Usefulness of User Interface Test Automation of HTML5-Based Hybrid Mobile Applications: A Case Study

Markus Kokkonen

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.
Espoo 6.5.2019

Supervisor

Prof. Casper Lassenius

Advisor

MSc Aki Hiisilä

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Markus Kokkonen

Title Usefulness of User Interface Test Automation of HTML5-Based Hybrid Mobile Applications: A Case Study

Degree programme Master's Programme in Computer, Communication and Information Sciences

Major Software and Service Engineering

Supervisor Prof. Casper Lassenius

Advisor MSc Aki Hiisilä

Date 6.5.2019 **Number of pages** 68 **Language** English

Abstract

While the first research papers on GUI test automation date back to the 1990s, its use in the software industry is still relatively rare. Traditionally, GUIs have been tested manually only and software automation has focused on the lower levels of testing such as unit testing or integration testing. The main reason for that is the complexity of GUIs compared to the lower software components. However, the manual testing of GUIs is a tedious and time-consuming process that involves repetitive and dull tasks, since the same tests need to be executed repeatedly on every testing iteration of the software under testing.

The main goal with GUI test automation is to automate those steps and by doing so improve the cost-efficiency of the testing process and free the testers' time on more meaningful and useful tasks. The previous research on GUI test automation reveals contradicting results. Some of the research has found GUI test automation to be both beneficial and cost-efficient and while others have found results suggesting the exact opposite.

The contradicting results from previous research and the unclarity on the benefits, challenges, limitations and impediments of GUI test automation worked as the main driver for this thesis. The research was conducted as a combination of a literature review on the subject and a case study of three HTML5-based hybrid mobile application projects in the mobile development unit of one of the biggest IT companies in Finland.

Keywords user interface testing, test automation, regression testing, test cases as documentation, structural testing

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Markus Kokkonen

Työn nimi HTML5-pohjaisten hybridimobiilisovellusten käyttöliittymätestauksen automatisoinnin hyödyllisyys: Tapaustutkimus

Koulutusohjelma Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

Pääaine Ohjelmisto- ja palvelutuotanto **Pääaineen koodi** SCI3043

Työn valvoja Prof. Casper Lassenius

Työn ohjaaja MSc Aki Hiisilä

Päivämäärä 6.5.2019 **Sivumäärä** 68 **Kieli** Englanti

Tiivistelmä

Käyttöliittymätestiautomaation käyttö yrityksissä on verrattain harvinaista, vaikka ensimmäiset tutkimukset aiheesta ovat 1990-luvulta. Perinteisesti käyttöliittymiä on testattu manuaalisesti ja ohjelmistotestausautomaatio keskitetty ohjelmiston alempien tasojen testaamiseen yksikkö- ja integraatiotesteillä. Pääsyy tähän on käyttöliittymien monimutkaisuus verrattuna alemman tason ohjelmistokomponentteihin. Käyttöliittymän testaaminen manuaalisesti on kuitenkin vaivalloinen, aikaa vaativa ja toisteinen prosessi, koska samat testit suoritetaan jokaisella testiajolla.

Käyttöliittymätestauksen automatisoinnin päätavoite on testauksen kustannustehokkuuden parantaminen ja testaajien ajan vapauttaminen olennaisempiin tehtäviin. Aikaisemmat tutkimustulokset käyttöliittymätestiautomaatioon liittyen ovat ristiriitaisia. Osa tutkimuksista on todennut käyttöliittymätestiautomaation olevan hyödyllistä ja kustannustehokasta ja osa on päätynyt päinvastaisiin tuloksiin.

Tämän työn päämotivaattoreina toimivat aiemman tutkimuksen ristiriitaiset tulokset ja epäselvyys käyttöliittymätestiautomaation hyödyllisyydestä ja kustannustehokkuudesta. Työn päätavoitteena oli tutkia voiko käyttöliittymätestiautomaation käyttö olla hyödyllistä ja kustannustehokasta. Työ koostuu kirjallisuuskatsauksesta ja kolmen HTML5-pohjaisen hybridimobiilisovelluksen tapaustutkimuksesta testiautomaation hyödyllisyyteen ja kustannustehokkuuteen liittyen.

Avainsanat käyttöliittymätestaus, testiautomaatio, regressiotestaus, testitapaukset dokumentaationa, rakennepohjainen testaus

1 Introduction

Software plays an important role in our everyday lives. The majority of the most popular applications are either web or mobile applications or mobile web applications. All of them have a graphical user interface (GUI), which works as a two-way messenger between the user and the underlying systems. The GUI displays the relevant information for the user and user actions and inputs manipulate the data and the state of the backend systems behind the GUI. For any software to succeed a certain level of quality needs to be delivered. A software with defects can be both annoying and harmful. While true for all software, quality control and testing is an important part of the development process for GUIs as well. In a typical software development project testing can take over half of the total time of the development process (Sharma and Angmo, 2014).

Traditionally GUI testing has been performed only manually and automation only applied for other parts of the system or lower levels of the software such as unit testing on the class and function level and other parts of the system or integration testing for a web application REST API (Engström and Runeson, 2010). GUI test automation isn't a new subject, but it still isn't as widely used a practice in the industry as other types of test automation. The main hindrance for GUI test automation to become more popular are the costs and the effort associated with it.

All approaches to GUI test automation require a significant upfront investment for implementing them and the tests need to be kept up to date with the changing software for them to stay useful (Carvalho, 2016). However, all current manual practices for GUI testing require a significant amount of time throughout the whole lifecycle of the software as well. One of the most common expectations related to GUI test automation is that the reduced time needed for manual testing exceeds the time required for the implementation and maintenance of the automation (Alégroth et al., 2015).

The aim of this study is to evaluate the usefulness and cost-effectiveness of GUI test automation and the factors affecting them through a literature review and a case study. The literature review presents different GUI test automation techniques and discusses benefits, cost-efficiency, challenges, limitations and impediments of GUI test automation. The case study involves three case projects that have no existing test automation and no existing testing processes apart from ad-hoc testing conducted as a part of the development process.

The initial goal of the case study was to create a test automation plan, design and implement GUI test automation and collect data from the use and maintenance of the tests when the development team adopts test automation practices as part of their development processes. However, despite the will and a plan to do so, the development teams haven't yet started using any of the implemented test automation for the case projects. As the answer wasn't just that the team didn't want to start using GUI test automation, six of the development team members were interviewed to gather data on the reasons and impediments for the adoption of test automation as part of their development processes. As the end result, there is only data and results on the implementation of the test automation and the impediments of its adoption, but none on its use and maintenance. This shifted the focus of this research from the original goal of purely evaluating the usefulness and cost-efficiency of test automation towards the impediments and challenges of its adoption and use.

The research questions for this thesis were formed based on the above-mentioned research goals:

1. Can the use of GUI test automation be considered beneficial and cost-efficient and which are the main factors affecting this?
2. What are the typical impediments to and challenges of adopting and using GUI test automation?

This thesis starts with the presentation of the results from the literature review (section 2). The literature review is followed by the case description (section 3) and the description of the methodology (section 4) used in this research. The next two sections present the results from the implementation of the test automation for the case projects (section 5) and the collected data on its usefulness and cost-efficiency paired with the interview results (section 6). The final section discusses the results and presents the conclusions based on them (section 7).

2 Literature

This section presents previous research on the subject of test automation. GUI test automation is typically used and best suited for regression testing so chapter 2.1 presents relevant background information on it. The next chapter 2.2 introduces the subjects of GUI testing, GUI test automation and different types of GUI test automation. Chapter 2.3 presents a comparison of manual and automated testing. The final two chapters 2.4 and 2.5 of this section discuss the benefits, limitations, challenges, impediments and cost-efficiency of GUI test automation.

2.1 Regression testing

Software testing is a wide term that describes various types of activities and tasks that aim to find defects, verify that the functionality conforms to the specification and measure the quality of software (Whittaker, 2000). The bigger the code base of a piece of software grows, the harder it gets to fully understand how a change in the code affects its behavior. Software testing is needed to ensure that the changes made work as intended and don't have any side effects. Regression testing is a term that groups together repetitive software testing practices that strive to ensure that changes to the software don't break any previously working parts in the software (Engström and Runeson, 2010).

Even in smaller scale-software development projects it can be hard to predict if a change affects the functionality in the desired way. In software development, bigger software size increases complexity and it can be impossible to fully understand every detail of the software under development. The larger a code base grows, the higher the probability of changes introducing new defects and causing undesired side effects grows. Instead of finding new defects, the main purpose of regression testing is to provide a safety net for making changes, i.e. catching these side effects as early as possible and preventing earlier issues from coming back. In short, well-planned and well-executed regression testing makes it safer to make changes. (Whittaker, 2000)

Regression testing can be done in any phase of the software development process. When conducted in an early phase, the goal is typically to catch defects as early as possible. And when conducted later in the process, the emphasis is more on validating the functionality (Engström and Runeson, 2010). In general, to keep the development process efficient and the feedback loop short, software testing should be planned so that it catches the defects as early as possible. Testing in the later phases of the development process mainly serves as certification and validation for the implemented functionality and not too many defects should be found that late.

The term regression doesn't restrict the type of testing. Anything from unit testing to UI black box testing to integration testing can be classified as regression testing as long as it's performed regularly over and over again on existing functionality.

2.2 GUI testing

Graphical User Interfaces (GUIs) are the standard way of interacting with any type of software nowadays and it has been so for decades now. They are an integral part of almost all software that people use daily such as mainstream desktop and mobile operating systems, the majority of desktop applications and all web and mobile applications. Therefore ensuring GUI quality is just as important as or even more important than ensuring the quality of any other parts of the software. This means that GUI testing is a relevant subject for most software development teams and projects and should be an important part of the software development process.

GUI Testing has also captured the attention of many researchers in the field. There's plenty of research on the effectiveness of different GUI testing techniques and tools (Brooks et al., 2009). In the past decade, the number of research articles on GUI testing has increased significantly (Banerjee et al., 2013). However, the terminology and categorization of the concepts isn't unified across this field of research. In 2013 Banerjee et al. conducted a systematic literature review on studies on GUI testing and they found that there is a need for improving reporting standards. Many research articles discuss and describe similar types of testing techniques and tools with different terminology. They also identified that the majority of the research articles on GUI testing focus on representing and evaluating new GUI testing techniques and usually involve test automation (Banerjee et al., 2013).

Generally, the purpose and the goals of GUI testing are the same as testing any other type of software. The goal is to deliver better quality by catching bugs and making sure the software meets the requirements specification, i.e. works as expected (Sharma and Angmo, 2014). Despite all the interaction and checks that happen on the GUI level, GUI testing is an end-to-end testing technique and it's possible to find defects in the underlying systems as well (Brooks et al., 2009). However, user interfaces and their program code is typically very different from other types of software (Carvalho, 2016). GUIs are very complex: there are multiple patterns of actions for users to accomplish the same end results. Finding the relevant parts and patterns to test is not a simple task and planning and executing efficient and comprehensive GUI tests is a time-consuming task compared to testing other parts of software. The majority GUI testing techniques are conducted on a very high level of system abstraction, which means that the focus is on measuring the SUT's conformance to the requirements. The complex nature of GUIs and testing on a high system abstraction level leads to larger and more complex test cases. This also makes test automation more challenging (Alégroth et al., 2015).

Selecting the correct GUI testing strategy is important. Berner et al. (Berner et al., 2005) suggests that an effective testing strategy doesn't rely on a single approach, but combines both manual and automated testing on different levels of abstraction, and that the strategy should be continuously evaluated and adapted. For example automated GUI testing is typically best suited for regression testing (Kasurinen et al., 2010). Manual testing should be used where GUI automation isn't feasible or possible such as with confirming certain non-functional requirements like conformance to design, usability or performance.

2.2.1 GUI test automation

Conventionally, test automation has been practiced and been most successful in unit testing and integration testing to verify the correct functionality of lower level system components. However, testing software on lower system abstraction levels needs to be complemented with higher level testing too to guarantee the quality of the SUT (system under testing) as a whole. Regardless of this, GUI test automation isn't as widely practiced as automation of lower level tests and GUI testing is still primarily a manual effort in many teams and companies (Alégroth and Feldt, 2017). GUI test automation isn't a new concept and it has been around in different forms for several decades already. The earliest research papers on GUI test automation date back to the 1990s (Brooks et al., 2009; Leivo, 2017). During the past few decades the tools and techniques for automating GUI testing have developed significantly and new ones are still constantly developed.

The main reason restricting more software development teams adopting GUI test automation techniques is that automating GUI testing is a hard and time-consuming task with no guarantees of it helping with deliver better quality with better efficiency than manual testing. User interfaces can be in a very high number of states and the number execution paths of user actions can be close to unlimited and they typically scale exponentially with the size of the SUT. Because of this any manual or automated approach for GUI testing is a time consuming task (Carvalho, 2016). While the complex nature of UIs makes automating the testing process such a hard task, it's also one of the main reasons behind the motivation for implementing it. The research about it confirms that manual testing is guaranteed to be a tedious process and test automation can have several benefits over it if used successfully.

At the core of all GUI testing is the monitoring of the changes in the GUI state as the result of the executed interactions and events through the GUI. Most GUI testing techniques are based on test cases that consist of chains of actions and events and expected GUI changes based on them (Carvalho, 2016). The test cases can either be manually written and programmed or generated based on a model of the SUT's GUI interactions and state transitions, but the principle is the same in both approaches.

2.2.2 Types of GUI test automation

There are several approaches to conducting GUI testing. On the top level, all software testing including GUI testing techniques is typically categorized into white box and black box testing (Liu and Kuan Tan, 2009). Black box testing, also called functional testing, describes all testing techniques that rely only on observing the software from the outside without any knowledge of the source code or internal functions of the software. Black box tests are based on the specification of the software and are designed around given inputs and received outputs and observed behavior of the software. White box testing, which also goes by the names of structural testing and glass box testing, is an umbrella term for all testing techniques that observe the software from the inside. White box tests are based on knowledge of the source code and focus on the internal functionalities of the software such as control and data flows (Nidhra, 2012).

Another common parameter for classifying testing methods is the level of system abstraction the testing focuses on. Both Vesikkala (Vesikkala, 2014) and Honkanen (Honkanen, 2016) used this classification method to divide testing methods into module / unit testing, integration testing and system testing. On the lowest level of abstraction, we have module level testing, which focuses on single functions and class methods. Integration testing is the next level from module level testing. It involves testing module groups and their compatibility and interoperability. On the highest abstraction level there is system testing. It's the category for all types of testing where the software is tested in its real environment; i.e. testing the whole system with all of its integrations and environment restrictions. As we can see in Table 1, the majority of GUI testing techniques focus on black box testing and either integration or system level testing.

As the terminology and classification for GUI testing methods varies greatly in different studies, I compiled the list for this research from multiple sources. Leivo (Leivo, 2017) and Carvalho (Carvalho, 2016) both listed and described different types of GUI testing in their papers. By combining the two lists, I got a list that covers all types I encountered in the studies examined for this literature review. Table 1 summarizes all these types of GUI testing identified. Carvalho categorized UI testing techniques to unit testing, capture / replay testing, random / monkey testing, model-based testing and pattern-based testing. Pattern based testing is a sub-category of model-based testing, so I grouped them together in this study. Leivo's division of categories was slightly different. He listed capture / replay testing, structure-based testing, visual GUI testing and model-based testing as different methods for GUI testing (Leivo, 2017).

Type of GUI testing	**White box / black box**	**Level**	**Principle**
Unit testing	White box	Component / Integration	Test single classes and methods at code level.
Capture / replay testing	Black box	Integration / System	Automate regression testing by recording manual test case runs.
Random / monkey testing	Black box	Integration / System	Find crash points, memory leaks, etc. with random inputs and other interactions with the GUI.
Model / pattern-based testing	Black box	Integration / System	Create a model of the UI and its behavior and generate automated test cases using the model.
Structure-based testing	Black box	Integration / System	Write automated test scripts that use the UI structure such as DOM for web pages to interact with and verify the UI state.
Visual GUI testing	Black box	Integration / System	Write automated test scripts that use image recognition to identify and interact with UI elements.

Table 1: Types of GUI testing

The methods for interacting with the GUI and interpreting its state can be divided into three main categories: coordinate-based, structure-based and visual. The coordinate based approach is one of the simplest and also the oldest approach. The test execution code and tool has no information about the structure of the UI and all of the interactions are based on the predefined GUI element screen coordinate locations. Even the slightest change in the UI style or structure can break tests based on screen coordinates, so this approach has widely been replaced by the other two. The structure-based approach relies on the representation of the GUI structure. All of the test case actions, events and checks are applied to elements in the structure. Websites are the typical environment where this approach is applied. In web apps everything that the user can see on the screen is based on the state of the DOM tree, which a representation of the structure and state of all of the elements in the GUI. The visual approach is based on advanced image recognition techniques and a high level model representation of the GUI structure and states. In a way it’s just a more advanced version of the structure-based approach for types of software where the representation of the GUI structure isn’t otherwise available or is needlessly complex. These techniques are covered in more detail in the chapters below, which introduce different types of GUI testing.

2.2.3 Unit testing

Unit testing is one of the most common testing techniques. Unit testing is a white box testing technique that verifies the functionality of single classes and functions by checking that the resulting outputs match the expected results with certain inputs. It’s the only GUI testing technique that uses the white box approach and focuses on lower system abstraction levels.

Unit testing can be considered to be the foundation of testing and test automation. It allows the developer of the tests to focus on the building blocks of the software. Focusing on one module at a time keeps the testing conditions simple compared to testing on higher levels of system abstraction. The simple conditions help to identify incorrect behavior and narrow down problems faster. Another benefit of unit testing is that the tests can be run often, because running them is relatively fast as they can be run in parallel (Vesikkala, 2014). Also, typically Unit testing doesn’t require complicated testing setups and environments, which means that they are simpler and faster to make part of the continuous integration process.

Due to the typical characteristics of GUI program code, a fair share of the code is really hard to cover with unit tests. Unit testing is solely based on giving a module certain inputs and then comparing the resulting outputs to the expected outputs. The display and user interaction logic of the UI layers of applications typically doesn’t produce any easily measurable outputs on the module level as they only alter what’s visible on the screen.

It’s possible to apply unit testing for higher level components, but in practice it isn’t feasible, because applying lower level testing techniques on higher level components leads to too much complexity, decreased maintainability and increased costs of testing (Alégroth et al., 2015).

2.2.4 Capture / replay testing

Capture / replay testing is a black box regression testing method that involves generating test scripts by recording manual execution of test cases. The recording process is identical to the manual execution of test cases with the only exception being that a recording software is used to capture interactions with the UI. Depending on the tool used, the interactions with GUI elements can be captured either as screen coordinates or references to the elements on the GUI structure level, such as the DOM tree with web applications (Carvalho, 2016). Typically, the recording software records user interactions as scripts. The scripts can be used as is or modified by adding assertions to verify expected UI behavior and states (Leivo, 2017).

Leotta et al. (Leotta et al., 2013b) compared the costs of setting up and maintaining capture / replay tests and programmable web testing. Compared to other system level white box testing techniques, the initial effort required to create and set up capture / replay tests is much lower. However, the maintenance costs of them is significantly higher. They found that only after a couple of releases were the costs of capture / replay tests higher than with programmable tests. The explanation for the lower cost of setting up capture / replay tests is simple: recording test cases is a simple and straightforward process compared to writing fairly complex test scripts. The reason for higher maintenance costs is that even a slightest change in the GUI can lead to tests breaking. While manually written test scripts can simply be updated for the part that has changed, the recorded scripts need to be completely re-recorded. The cost of creating versus maintaining capture / replay scripts is close to constant.

Another aspect to comparing capture / replay tests with any programmable tests are the skills required for using the testing techniques. For effective use, all of the techniques require basic skills required in all software testing such as designing the test cases, test data generation, organization and usage, knowledge on how to effectively test both functional requirements (FRs) and non-functional requirements (NFRs), and so on (Dustin, 2002). However, contrary to any programmable testing approach, setting up and running capture / replay tests only a very basic level of programming skills or none at all (Leotta et al., 2013b).

Carvalho (Carvalho, 2016) also points out that the recording process can be rather tedious and prone to errors. Making an error in recording the test case means that the recording needs to be restarted. Also, when the tests are re-recorded, there is a window for human error and the recording might differ from the previous recording for the same test case. These points are true especially for longer test cases.

2.2.5 Random / monkey testing

Random / monkey testing is a black box testing technique where random inputs and user interactions are executed to find memory leaks and program crashes. The purpose is not to test anything specific, but to find unexpected behavior and crash points with completely random inputs.

There are some serious limitations to what kind of problems random / monkey testing can detect. The automation tool doesn't know anything about the software state or the requirements, so it's impossible to detect any deviations from the requirements specification in the functionality or detect errors that are handled correctly in the UI. Also, since the problems are found as the result of a long sequence of random inputs, it can be really hard to figure out how to reproduce the problem. (Carvalho, 2016)

While random / monkey testing can help find some defects not easily detectable by other testing techniques, because of its limitations it can never replace any other type of testing. It also seems that it isn't a very widely researched or used testing practice in the industry. The majority of the articles studied for this literature review didn't mention this testing technique at all or only mentioned it as a side note.

2.2.6 Model / pattern-based testing

Model-based testing is a testing approach where the first step is to create a model of the software, its states and state transitions and then automatically generate the test cases from the model. The model is created based on the requirements specification and it represents what the software should be and how it should behave when interacting with it. When applied to GUI testing, the model describes the relevant UI views and elements and their states, the interactions with them and the transitions between the states (Silva et al., 2008). The models are generally divided into three categories: graph models, operational models and behavioral models. All of them describe the UI states, their relations and transitions between them, but from a different perspective. For example, operational models approach the problem from the perspective of typical real-life use cases, where graph models just describe the UI states and transitions between them (Brooks et al., 2009). Pattern-based GUI testing is a sub-category of model-based testing using pre-defined general behavioral elements to model the GUI (Carvalho, 2016).

In addition to creating the model of the UI, the model needs to be mapped to the actual UI elements and events. Without the mapping, the automation tool has no way to run the generated tests on the actual UI. Another challenge that needs to be tackled is finding the sufficient, but not too detailed, abstraction level for the model. With too high level of abstraction, the model doesn't describe the actual software with a sufficient level of detail and the test cases derived from the model aren't sufficient for verifying conformance to the requirements. With too much detail, the possible number of states of the model along with the possible number of generated test cases grow to approach infinity. Some of the model-based testing tools offer possibilities to guide the test case generation process to help keep the number manageable (Silva et al., 2008).

Regardless of the limitations of the testing technique and the vast amount of work required to create the models, model-based testing is one of the few GUI testing techniques capable of finding new defects outside of the test automation development phase. Most other types of testing techniques mostly catch the issues during their development and after that the tests serve as regression tests, i.e. a safety net for detecting issues coming back or new ones arising in existing functionality covered by the tests. The enabler for this difference is the dynamic test case generation based on the models. The dynamic generation makes it possible to find completely new issues when the models are changed or the generation parameters tweaked (Brooks et al., 2009). Also if the models are designed on a level of sufficient detail, it's easier to achieve higher UI state coverage compared to types of testing where test cases are created manually (Sivanandan and B, 2014).

2.2.7 Structure-based testing

Structure-based testing, also known as programmable testing, is based the on manual creation of test cases. The structure based testing tools and programming rely on a structural representation of the UI. This approach is typically used in web development where the DOM tree represents the structure and the state of the application. In modern web applications, instead of full page reloads, the application uses asynchronous data fetching and DOM tree manipulation to deliver improved performance and a more seamless experience on mobile devices. In programmable structure based tests, the test cases are programmed as sequences of actions performed on the UI elements and checks and verifications performed on their states. In the context of web applications, this could mean for example performing clicks on button elements and verifying the expected state changes with assertions on certain element values (Leotta et al., 2014). One of the possible benefits achieved with this testing approach is that the test cases can be derived from previously defined manual test cases, since the test cases and their execution are very much like manual test case executions with the difference being the human tester is replaced with a programmed test script. Another benefit in the structure-based approach is that it doesn't depend on the visual style or shape of the UI objects but the underlying structure instead. The whole application can be themed or styled differently and it doesn't affect the tests in any way.

Structure-based testing has the same obvious drawback as any programmable test automation. Writing and maintaining the tests requires manual work. The initial effort of setting up and implementing test automation applies to all approaches, but keeping programmable tests up to date with the changes in the SUT can require a significant amount of work.

Using page objects is one of the best-known practices for structure-based testing. The core idea is to represent the UI views with page objects. A page object encapsulates the possible actions the user can perform in the view and provides access to the relevant UI elements in the view. The page objects work as a layer of abstraction between the UI structure and the test cases. In a way, the page objects can be seen as an API layer between the test case and the raw UI structure. When writing the test cases, the test case programmer can use the page object layer to interact with the UI and access its state. The main goal with using the page object approach is to improve code reusability and make the test case code simpler and easier to understand and less cumbersome to write. A page object can for example provide simple functions for performing chains of actions and checks needed in several test cases. Instead of copying and pasting the code for the actions, the programmer can simply use the functionality provided by the page object to perform the desired actions and UI state checks (Leotta et al., 2013a). Also, when the SUT changes, the layer of abstraction provided by the page objects can help restrain the work needed for updating the tests. When the changes only affect the functionality that the page objects already represent and the functionality doesn't significantly change, only the page object needs to be changed and none of the tests using it need to be touched.

2.2.8 Visual GUI testing

Visual GUI testing (VGT) is one of the newest GUI testing techniques. The main difference from other types of GUI testing is that it uses image recognition to interact with the UI. The tests are written as scripts and one of the several available tools is used to automate the tests. (Alégroth et al., 2015)

One of the biggest advantages of VGT is that it doesn't rely on the underlying technologies or platforms, since the interaction with the UI is based on image recognition. This is similar to a human testing the software manually and using their eyes as the main source of information for recognizing and interacting with UI elements. Otherwise VGT-based tests and their benefits are very similar to the benefits of structure-based testing as the main difference between them is the means of recognizing and interacting with UI elements.

Visual GUI testing has its limitations and challenges. In their case study on the subject, Alégroth et al. categorized their findings of limitations and challenges into three categories: issues related to the SUT, the test tools and the support software. The majority of the issues were related to the SUT and included issues such as the tested version of the software not matching the test cases and specification, image recognition timing issues with non-instant state transitions, irrelevant defects encountered in the middle of a test run terminating the test execution, and all system features not being available in the testing environment due to missing integrations. However, the most significant issues were related to the testing tools used and included failures in image recognition, corrupted images, missing tool documentation and functionality. Despite these challenges, the study concluded that the participants of the case study found automated to be valuable and cost-effective and at least as good at finding defects as manual tests would have been. (Alégroth et al., 2015)

2.3 Manual vs automated GUI testing

Software testing takes on average from 40% up to 70% of the time of the whole development process (Sharma and Angmo, 2014). Traditionally, most of this time has been used for manual testing. In their 2010 survey on regression testing practices Engström et al. found that the industry is divided on the usefulness and cost-effectiveness of test automation and that half of the respondents used as much manual and automated testing equally much. Also, 60% of the respondents were happy with their manual testing and 30% used no test automation (Engström and Runeson, 2010).

The main incentive for the transition from manual GUI testing towards automation is reducing the costs and time required for the whole testing process while maintaining or improving the effectiveness of testing. Despite the goal of saving manual testing time, the main goal of test automation shouldn't be replacing manual testing altogether. Test automation is the most suitable for regression testing with the goal of ensuring that existing functionality still works after changes are made (Kasurinen et al., 2010). With manual regression testing it's important to carefully choose the frequency of the test runs an which test cases are selected for each run (Engström and Runeson, 2010). Since the execution of the tests takes close to no manual effort, the introduction of automation decreases the relevance of those decisions as all the tests can be run frequently. However, the more relevant question with automation is which test cases should be automated and which ones run manually.

Test automation is rarely able to find new defects or defects it isn't designed to find (Rafi et al., 2012). Because of this, GUI test automation can help reduce the workload of manual testing, but manual testing is still needed for verifying functionality that isn't feasible or possible to test automatically such as very complex test cases, animations or usability, performance and other typical NFRs.

According to Alégroth et al. one of the main reasons that manual testing is still widely practiced over test automation is that the currently available GUI test automation techniques and tools aren't flexible and cost-efficient enough (Alégroth et al., 2015). Different types of applications, such as desktop apps, web apps and mobile apps, have very different types of UIs implemented with various technologies. Each of them have their own tools for test automation and typically several different automation tools and technologies are available for the same types of UIs. Because of the limited applicability of the automation tools and the fact that GUI test automation isn't very widely used in the industry, the tools are limited and have a narrow scope of applicability and haven't yet matured to their full potential. One of the findings by Kasurinen et al. (Kasurinen et al., 2010) was that many companies and teams end up developing their own tools for test automation because of the aforementioned reasons.

2.4 Benefits and challenges of GUI test automation

The drawbacks of manual testing and the possible benefits of test automation have led to an increasing amount of research on the subject. Table 2 shows a list of common benefits and Table 3 challenges, limitations and impediments of GUI test automation. The lists were compiled from several sources. The list was formed based on the results of a systematic literature review study of the benefits and limitations of all types of test automation by Rafi et al. (Rafi et al., 2012) The initial list was complemented with benefits and limitations specific to GUI testing with results from studies conducted by Alégroth et al. (Alégroth et al., 2015; Alégroth and Feldt, 2017), Berner et al. (Berner et al., 2005), Wiklund et al. (Wiklund et al., 2017) and Kasurinen et al. (Kasurinen et al., 2010). Many of the aforementioned studies listed most of the benefits and limitations and challenges listed in Table 2 and Table 3, so the results from the different studies were very overlapping.

Benefits of GUI test automation
1. Reduce overall development costs
2. Reduce testing effort
3. Improve software quality: increased fault detection
4. Make testing less mundane for the tester: eliminate tedious parts of manual testing
5. Improve test reliability: Eliminate error-prone parts of manual testing
6. Enable more frequent execution of the tests: faults detected earlier in the development process
7. Improve perceived quality & confidence in the system

Table 2: Benefits of GUI test automation

Challenges, limitations and impediments of GUI test automation
1. High effort and costs: design, implementation and maintenance
2. Automation tools have a high learning curve and limited context they're applicable at
3. Ineffective / incorrect testing strategy: which types of testing and which test cases should be covered by automation?
4. Lack of required skills and knowledge related to the tools, programming, domain and the SUT requirements
5. Unrealistic expectations: unmet high expectations decrease perceived value of automation
6. Organizational challenges: business decisions prevent test automation to be adopted
7. Challenges adapting testing processes: automation processes need time to mature and setting up automation infrastructure requires time and effort

Table 3: Challenges, limitations and impediments of GUI test automation

It's worth noting that most of the studies used for gathering the data for the tables above don't focus specifically on GUI test automation, but test automation in general or focus on a specific approach for GUI test automation, but the lists were compiled based on research articles' findings applicability for GUI test automation.

2.4.1 Benefits of GUI test automation

Usually one of the main motivators behind GUI test automation is the same as with all test automation, which is to increase return on investment (ROI) and resource use efficiency for the testing process. According to Garousi et al. test automation usually reduces testing effort (Garousi et al., 2013). However, for various reasons, the implementation and maintenance costs are very high and can exceed the costs saved in manual testing (Rafi et al., 2012). The cost-efficiency of GUI test automation is discussed in more detail in chapter 2.5.

In addition to reduced development costs, the decreased effort required for test execution has other benefits too (Rafi et al., 2012). Because of the time-consuming and tedious nature of manual testing, test execution can't be carried out very often, typically once a week, or as part of the version release process. As a result of this, the issues and bugs are found a long time after a change in the SUT was implemented. Ideally, when the GUI tests are automated and can be run in a shorter time and with close to zero effort, they can be executed often and possibly along with all other test automation close to or as a part of the software development cycle. There's no universal model for how the testing process is tied to the rest of the software development process, but having test execution close to the development cycle is one of the cornerstones of modern fast-paced agile software development. This enables a faster feedback loop between the development cycle and the testing process.

The fast feedback loop has several benefits. Issues caught by the tests can be fixed right away, possibly even as part of the development process. As a result of this, all changes to the SUT have already been tested with high test coverage during the software development cycle and ideally the SUT should always be ready for the next release (Rafi et al., 2012). Also, when the issues are already caught during the implementation phase, the developer remembers all relevant requirements and code related to the task. This might not be the case if the issues are found a couple of weeks later when manual tests are carried out. All in all, the faster feedback loop enables faster software development with higher confidence for better quality delivered together with frequent releases.

Test automation can also help improve test quality (Berner et al., 2005) and ensure consistent and reliable test execution (Rafi et al., 2012). Test automation can help reach higher testing coverage thanks to the greatly reduced manual work required for test execution. Large and detailed test suites might not be feasible or cost-efficient to execute manually over and over again. Another edge test automation has over manual testing is that manual testing is a tedious and repetitive process that is prone to human errors. The testers can focus their time and energy on more important tasks than manually executing the same tests over and over again. Also, test automation can be considered more reliable than manual testing. Often test cases require very specific data and sequences of steps and actions. The automated tests are guaranteed to perform the tests exactly the same way on every execution, while human testers are prone to errors or might get bored repeating the same steps and take shortcuts or make other deviations in the test execution (Wiklund et al., 2017). The higher test coverage and improved test reliability can increase the fault detection capabilities of the testing process.

In addition to and partly thanks to improved test quality, test automation can help improve product quality and perceived confidence in the SUT. A very common problem is that a change somewhere in one feature breaks something else that wasn't intended to be changed. The improved regression testing capabilities brought by test automation make it safer to make changes in the

SUT. The frequently run regression tests ensure that any features covered by the tests still work after a change (Kasurinen et al., 2010). In practice the safety of making changes means that more problems are caught earlier in the development process and also that the developers can feel safer and more confident making changes and don't need to fear introducing new bugs in the SUT. This is helpful for all developers, but especially for ones who are new to the project or for some other reason don't know the SUT and its requirements that well.

2.4.2 Limitations, challenges and impediments of GUI test automation

Even though the main goal with test automation is typically to save time and money, most of the challenges, limitations and impediments of GUI test automation are either directly or indirectly related to its costs and required effort. All known GUI test automation approaches require a substantial amount of effort for implementation and maintenance (Carvalho, 2016). Many of the problems listed in Table 3 are related to the costs and required effort of GUI test automation. This chapter focuses on the reasons behind the high cost and other challenges of GUI test automation and the cost-efficiency aspect is discussed in more detail in chapter 2.5.

The first prerequisite for adopting GUI test automation or any test automation practices in any product development team is to set up the test automation infrastructure and select and learn the tools used for automation. Both setting up the required infrastructure and learning the tools require a significant amount of time (Rafi et al., 2012). In their literature review on impediments for test automation Wiklund et al. (Wiklund et al., 2017) found that the test automation tools have a steep learning curve. This was also confirmed by the findings of Alégroth et al. (Alégroth et al., 2015) in their case study on the benefits and limitations of VGT. Also, due to the complex nature of GUIs, designing and implementing automated tests for them is more time-consuming than for simpler software components with a more narrow and specific functionality such as APIs or functions tested by unit tests.

Another relevant characteristic of the automation tools is their limited applicability. The tools can typically only be used with certain implementation technologies and have other severe limitations. Failure to take the technologies and processes into consideration when choosing a fitting tool can cause the adoption of test automation to fail altogether (Wiklund et al., 2017). For example, mobile or desktop application GUI test automation tools can't be used for testing web applications or vice versa. In many cases the factors affecting the tool suitability might not be as clear as the given example. Also, in many cases none of the available tools are suitable as is and many companies end up extending existing test automation tools or developing them from scratch (Kasurinen et al., 2010). The limited applicability of the tools also has the implication that the testers' knowledge and skills in using the tools is to some extent only applicable to projects with similar types of GUIs and that they have to start the learning process over when working with new types of GUIs.

Testability of the SUT is another typical limitation associated with test automation. If testability hasn't been taken into consideration from the beginning of the project, introducing test automation might be challenging (Wiklund et al., 2017). A good example of testability in the context of GUI test automation is the ability of the test case to verify system state changes during the test (Rafi et al., 2012). If the data in the underlying backend system for the GUI can't be reset on command for each test execution, any changing data imposes problems for verification of the results and consistent test execution without the use of mock data. Poor testability of the system can increase the costs and decrease the value and ROI of test automation (Wiklund et al., 2017).

A common challenge with using test automation is choosing the correct testing strategy (Rafi et al., 2012). Choosing a testing strategy involves choosing which testing approaches are used to test different parts of the system, which tools are used and which levels and parts of the SUT are covered by test automation and which by manual testing. According to Berner et al. (Berner et al., 2005) choosing the right testing strategy is one of the most important factors in successful use of test automation. It's important to recognize the limitations of test automation and recognize that automation can typically only find issues it is designed to find (Rafi et al., 2012). Also, the cost of implementation and maintenance for test case automation should be considered with more complex test cases. The best and most efficient result is achieved when the aforementioned aspects are considered and both manual testing and test automation are used for the purposes and tasks they suit the best. (Berner et al., 2005).

One more common challenge with adopting test automation is that the existing testing processes need to be adapted for it (Rafi et al., 2012). Changing any existing processes takes time and effort and introduction of test automation is no exception. Test automation typically changes at least the frequency of how often the tests are run, introduces new skill requirements and might change the responsibilities of the members of the development team. For example, if the development team has separate testers, depending on the chosen automation tools, the tool might require them to know programming. Also, the developers might not have worked much with testing previously and now they might be required to develop and run the automated tests. Regardless of the details, the introduction of test automation changes the team's ways of working and requires new skills from the team members. The adoption of the use of test automation into the team's processes can be even more challenging and require extra effort if the team doesn't have any previous knowledge and experience with test automation (Wiklund et al., 2017). Even with a team willing to adopt the new processes this means an investment of time and effort that could be used for something else if nothing was changed.

The effort and costs of maintaining the test automation is one of the most common challenges of using test automation. Challenges related to test automation maintenance were mentioned in all five research articles used for compiling Table 3. This claim is also confirmed by the findings of Vesikkala: "*We found that keeping tests up to date with the tested application is the most significant issue with the existing visual regression testing tools.*" (Vesikkala, 2014).

The challenges of test automation maintenance are true especially for GUI test automation as GUIs tend to change more often compared to other types of software components (Berner et al., 2005). Any change in the GUI means that the related tests need to be changed as well. Maintainability and reusability are things that can and should be taken into consideration in the design and implementation of the test automation. Kasurinen et al. (Kasurinen et al., 2010) suggest that there are two mutually exclusive options for implementing test automation: easy implementation or high maintainability and reusability. With easy implementation, the maintainability suffers, but the implementation costs are lower. When maintainability and reusability is prioritized, the implementation costs are higher, but maintenance costs lower. Considering that all of the reference articles suggest that maintenance is one of the most common pitfalls of test automation, the latter approach of promoting high maintainability and reusability is probably the better solution for successful and profitable test automation.

Another common risk related to test automation is unrealistically high expectations (Rafi et al., 2012). A common misconception with test automation is that it saves time and money right from the beginning when the reality is that profitability can typically be achieved over a longer span of time. Failure to manage the organizations expectations can lead to decisions to abandon the use of test automation too early when the expected ROI isn't achieved fast enough (Wiklund et al., 2017). Even if the false expectations don't lead to such drastic decisions, they can decrease the perceived value and affect future decision-making on use of test automation.

Other organizational issues related to test automation are prioritization of work with more immediate benefits such as implementation of new feature over the implementation and maintenance of test automation in projects with time pressures (Wiklund et al., 2017). These kinds of decisions are highly likely to have negative effects on the usefulness and cost-efficiency of test automation. The first effect is that when the test automation isn't maintained, existing tests break and become useless. If the use of test automation is only dropped for a while, the other likely issue is with the amount of work required to get the test automation up to date with the new and changed features. A quote from Berner et al. describes the issue well: "*Automated test suites have a strong tendency to be quite unforgiving when suspended and not run for a short or even very short time.*" (Berner et al., 2005). Alégroth et al. (Alégroth and Feldt, 2017) suggests that the maintenance of test automation should be integrated into the team's everyday development processes to ensure that the SUT and the tests won't get out of sync.

2.5 Cost-efficiency of GUI test automation

A common expectation with test automation is that the time and money invested in implementing and maintaining test automation pays itself back in decreased effort needed for manual work while maintaining the same level of testing or improving it. The most time-consuming part of manual testing is the test execution phase, which is almost completely eliminated or at least greatly reduced for the functionality covered by test automation. As established in the previous chapter, manual testing takes a considerable share of the total time of the whole software development process. Because of this, even a fairly small reduction in the time required for testing is noticeably reflected on the total product development time and costs (Alégroth et al., 2015).

Increased resource use efficiency and ROI of testing processes, one of the main motivators behind all test automation, is also one the most common reasons as to why GUI testing is not automated. As pointed out in chapter 2.4.2, the cost of implementing and maintaining test automation is one of the most common reasons as to why manual testing is preferred over test automation. The main question around the cost-efficiency of test automation is if can the costs of test automation can be lower than the money and time saved directly and indirectly.

All research articles on the benefits and limitations of test automation used for the comparison in chapter 2.4 seem to agree on the high implementation and maintenance costs, but there are clear contradictions in the findings and conclusions about the long-term cost-efficiency of GUI test automation. One obvious explanation for this is that despite sharing a common goal of assessing the cost-efficiency of test automation, there are no industry standards or other well-defined metrics or methods for conducting the research. When there is no standardized methodology for the research on the subject, the results can be guaranteed to vary too. On a higher level the idea is simple: estimate the costs of developing, maintaining and executing the tests, add them to the costs of setting up and learning the tools and finally compare the total sum with the saved time in other areas of the software development process including testing.

While estimating the direct costs and savings might be quite straightforward, factoring in all of the indirect savings and the value of certain benefits isn't a simple task. A good example of this is trying to estimate how much time or money is saved when a bug is found and fixed in the development process as the result of an automated test catching it instead of it being found in manual testing a couple of weeks later. Only one of these scenarios can take place and issues caught during the development phase aren't typically reported anywhere, but fixed right away. It's obvious that the results of academic research on the cost-efficiency of test automation have variation in the results on cost-efficiency as long as the used metrics vary. A few more examples of often overlooked benefits in test automation ROI calculations are the positive effects of shorter release cycles, more frequent test execution and that the testers can focus on more important areas of testing (Berner et al., 2005).

Certain characteristics in the processes, SUT and design can affect the cost-efficiency of test automation also. In their research article on the feasibility of test automation Berner et al. (Berner et al., 2005) suggest that the variation in the findings on cost-effectiveness is caused variation in the suitability of the selected testing strategy, the testability of the SUT architecture and the test automation tool and setup feasibility and quality. According to them if testability is taken into consideration in the system architecture design, a fitting testing strategy is selected and the right automation test tools are selected, test automation should be cost-effective. Alégroth et al. (Alégroth and Feldt, 2017) add to this by claiming that GUI test automation cost-efficiency is also affected by certain project characteristics. Long-term cost-efficiency can be more easily achieved in projects spanned over a long period of time and preferably with multiple development iterations. Another suggestion by them is that at the start of a project there are no full features to be tested and the GUI tends to change significantly in the early phases of a project. So to save on the maintenance costs, test automation shouldn’t be introduced from the start of the project, but after a certain maturity has been reached.

The results from previous studies on cost-efficiency of test automation vary. For example, with two case projects Alégroth et al. (Alégroth et al., 2015) confirms that GUI test automation can be applied cost-effectively with the VGT approach. The breakpoint for positive ROI was a bit below 15 executions. The 12 companies included in the study by Kasurinen et al. had contradicting results related to the cost-efficiency and usefulness of test automation (Kasurinen et al., 2010).

3 Case description

This section presents background information on the case projects. Chapters 3.1, 3.2 and 3.3 start with the introductions to the case projects, the development team working on them and their testing practices. Chapter 3.4 explains the motivation for implementing test automation for the case projects. Chapter 3.5 presents the technologies and architecture of the case projects. Chapters 3.6 and 3.7 explain the process of selecting the testing tools and principles and conventions set for designing, documenting and implementing the automated test cases.

3.1 Introduction to the case projects

The products selected for the case study are very similar to each other. All of them are HTML5-based hybrid mobile web applications. Also, all of the selected applications are fairly simple UIs for existing backend systems. All of the backend systems have been in production use for over a decade, have a wide variety of functionality and fairly complex UIs. The mobile development teams are from different OUs (organization units) than the OUs that own the main products. The OUs develop the mobile apps in co-operation with OUs where the domain knowledge and requirements come from the backend system OU and the mobile development expertise from the mobile development teams.

The main purpose of these mobile applications is to simplify and provide access to some of the core functionalities of those systems on the end users' mobile devices that have traditionally only been accessible only through the customers' intranet. The main motivation behind these projects have been to streamline the customer's business processes around the functionalities brought to the mobile applications. Additionally, compared to the traditional UIs, the mobile apps are able to provide faster and easier-to-use access to the frequently used core features of the backend systems.

Table 4 summarizes some relevant details about the case projects. Some of the projects are already in production use, but all of them are still under continuous development. All of them are business-to-business applications, i.e. they are not available to the wide public, but only to the customers of the backend system products. Their branch of business is determined by the backend system they're integrated to and the focus is different in each project. They are all HTML5 mobile web applications, but some of them have also been released as native Android and iOS hybrid applications. The hybrid applications for all platforms share the same web UI, but use the Cordova framework to access mobile OS level features otherwise inaccessible to web applications.

Project	Short description	Year started	In pro-duction	Size	Device support		Platfrom support		
					Mobile	Tablet	Web	Android	iOS
Project 1	View payslips	2016	Yes	S	x			x	x
Project 2	Hour logs, vacations, absences	2016	Yes	M	x	x	x		
Project 3	Driving logs, expense & travel invoices	2016	No	M	x	x	x	x	x

Table 4: Case projects

3.2 Team background

The mobile development team working on the projects was formed in 2014 and was initially a small Scrum team of 4 developers all based in Finland. At the moment, the development team consists of three Scrum teams with each having their own Product Owner, Scrum Master and eight to ten developers. Roughly half of the developers are based in Finland, a quarter in Estonia and the remaining quarter in India.

Due to the success of the development projects and a growing demand for mobile applications for more efficient, faster, more accessible and easier ways to use the backend system functionality, the team and its number of projects has grown steadily for 4 years now. The constant growth has required a constant change and improvement in the software development processes including design and documentation processes, development workflows, release processes and quality assurance. Initially, the processes were very loosely defined and lightweight. But as the team grew larger, the processes have developed into stricter and more accurately defined processes to ensure a predictable and steady rate of delivery of features and quality.

3.3 The current state of testing in the case projects

All of the products are web-based UIs with minimal business logic that communicate with the backend systems through a REST/JSON (JavaScript Object Notation) API. Because of this, most of the code base mainly consists of program code to either fetch / send data, manage the app state or UI views and their interactions and transitions. Due to the nature of the code base, a very small portion of it can be considered unit testable. Testing a feature against the requirements manually and reviewing all code changes has been part of the team's development workflow and Definition of Done since the beginning. However, test cases haven't been documented or designed for most of the case projects and testing practices outside of the quality assurance practices integrated into the software development workflow. Also testing process have varied from project to project. Many of the projects don't have any test cases designed or documented and the feature documentation for many of the features is lacking. The team's testing practices have remained almost unchanged while the team size has grown and the products become more complex.

Initially, when the team was smaller and the code base less complex, every developer in the team knew the features and the code base of the products under development thoroughly. This is not the case anymore, as many of the developers originally working on the projects aren't working on them anymore and the majority of the developers have jumped in mid-project. Adding new features and making changes in a complex and large code base and not knowing all of the features has a high risk of introducing unintended side effects and new defects in the existing functionality. The situation has led to an increasing number of bugs and other side-effects getting through the team's quality assurance processes. Certain problems have been fixed and come back multiple times.

3.4 Case motivation

The team discussed the arising problem described above on several occasions in the team's retrospective meetings. Based on these discussions, the team identified the need for improved requirements and feature documentation and heavier and more thorough testing processes. The team's consensus for improving the testing processes to solve the problem was to create and automate test cases with the main goal of preventing regressions.

Improved requirements and test case documentation should help the developers better understand the products they are working with. This knowledge should help them catch and fix regressions already while working on changes and new features.

Manual testing is cumbersome and time-consuming and the possibilities for spending more time on testing are limited. The team also wanted to keep the main testing feedback loop as close to the development process as possible with the goal of keeping the number of regressions and other defects getting to sprint releases minimal. With these requirements the decision to go with test automation seemed obvious. Test automation enables running even larger sets of tests often. Initially, the tests would be run manually as part of the feature testing process and eventually the tests would be a part of the continuous integration (CI) process.

The main goal with the test automation is to make it safer to make changes and free up time for more complex manual testing by eliminating the need to manually execute the same simple regression test cases over and over again. In short, the ultimate goal is to make the testing process more comprehensive and focused and to make better use of the available resources.

3.5 Technologies

3.5.1 The tech stack in the case projects

As already established in the case project introduction, all of the case applications are HTML5-based single-page web applications. The majority of the case project applications are also available as native hybrid applications on

Android and iOS mobile devices. The web applications are wrapped as native applications with the Cordova framework.

The web application core is built with the jQuery Mobile, Backbone.js and RequireJS. In addition, multiple utility JavaScript libraries are used for more specific common functionalities such as persistent storage access, language support, page templates and date and time handling.

All of the case project applications are single-page applications. This means that only one page is loaded at the application startup and all page transitions are handled by dynamically manipulating the page DOM tree. Also, all external data is fetched through a REST / JSON API provided by the backend system instead of fetching and showing pre-rendered pages.

The Cordova framework is used to package the web applications as hybrid native applications for Android and iOS platforms. Figure 1 shows the Cordova application architecture. The framework uses a simple native application that starts up the web application in the OS WebView.

The native OS functionality not available for regular web apps, such as sensor data and background processes, is accessed through a JavaScript API described in Figure 2. The main principle is very simple: The web applications calls an API function with success and failure callback functions as parameters which triggers the execution of the corresponding code on the native app side of the application. The native code then either calls the success or failure callback functions to return the response and any data to the web app side.

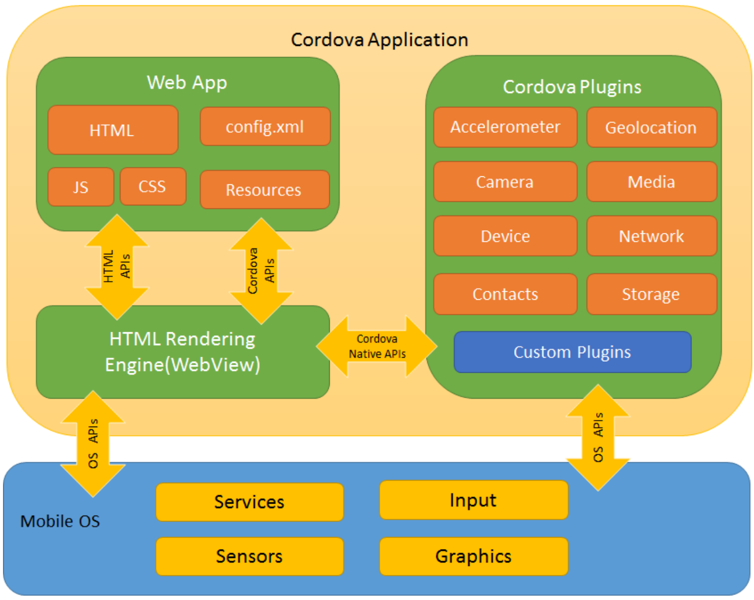


Figure 1: Cordova application architecture (The Apache Software Foundation, 2018)

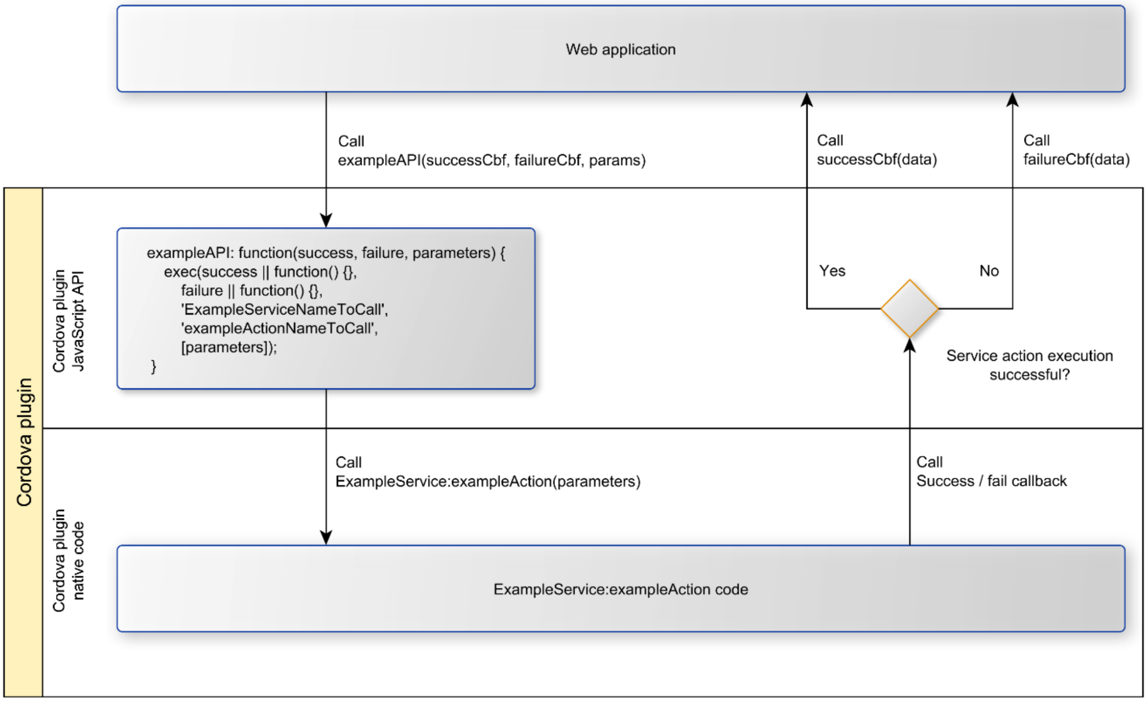


Figure 2: Cordova JavaScript API

3.5.2 Web application architecture

The web apps have a standard Model-View-Controller (MVC) web application architecture described by Figure 3. The MVC architecture model divides the application logic into three basic components with each having their specific responsibilities. The Model holds the application data and business logic related to it. The View is responsible for what is displayed for the user based on the Model. The controller handles inputs and signals from the View and updates the Model accordingly. Figure 4 shows how MVC reflects on the application architecture in practice. The data model is represented by the Models and Collections, the Router and the Presenters are the Controllers and the Views are responsible for rendering the UI based on the Model.

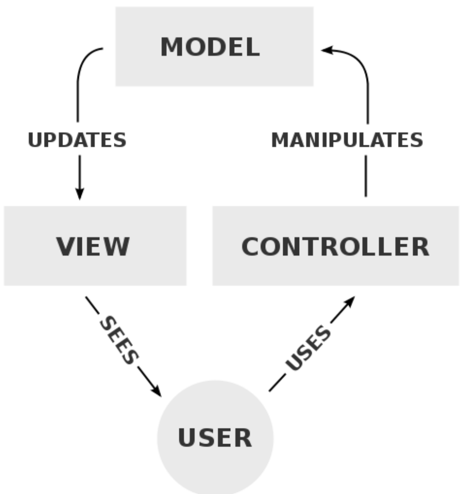


Figure 3: MVC architecture ("Model–view–controller," 2019)

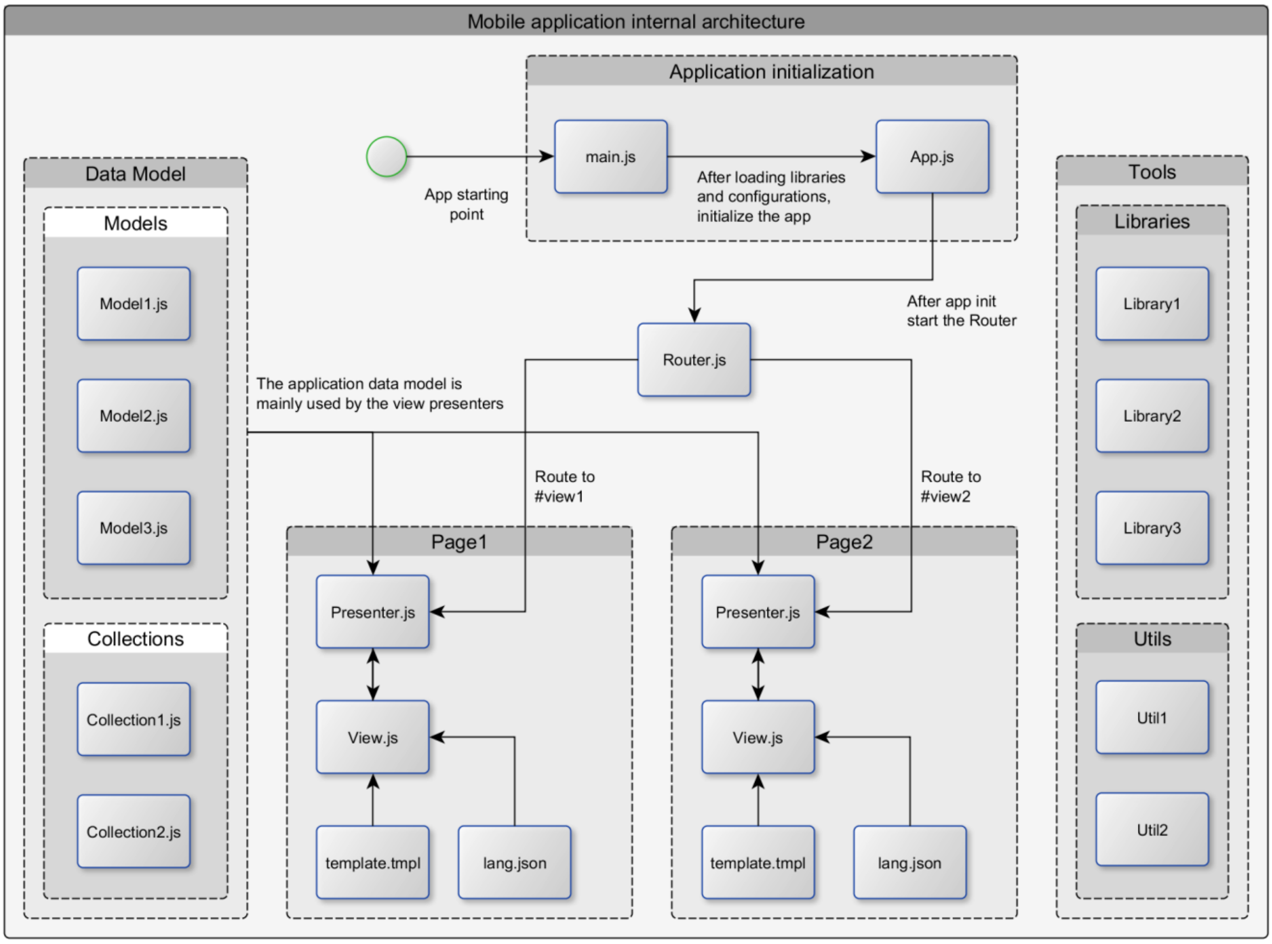


Figure 4: Case project application architecture

3.6 Selecting the test automation tools

The main requirements for selecting the test automation tools were set by the used technologies and the supported platforms. The automation tools had to support running the tests on desktop and mobile web browsers and also on the native Android and iOS web apps.

Selecting the tools for browser and native device automation was a straightforward process. The desktop browsers would be covered by the industry standard browser automation tool Selenium and the mobile browsers and hybrid applications by Appium. The team didn't find any serious alternatives to these tools. Also, they both support the WebDriver API standard for performing automated tests, which meant that the same tests could be run on the desktop browsers and the mobile platforms.

For the test framework a few alternatives were considered. Based on a short initial research process, the alternatives were narrowed down to Robot Framework, Nightwatch.js and Mocha. After a thorough comparison and trying out each of the frameworks, the team chose Nightwatch.js.

There are several reasons why Nightwatch.js was chosen. First of all, it uses the WebDriver API for browser automation tool communication, so the tests could be run both on Selenium and Appium. Figure 5 describes architecture of running the test in more detail. Second, Nightwatch.js is based on Node.js, which means the tests are written in JavaScript. All of the team's developers were already familiar with JavaScript, so the learning curve should be smaller than with a framework using some other programming language. Third, the tool seemed to be the most simple to set up and the most lightweight without lacking any essential features. Other benefits with Nightwatch.js are JUnit XML output for continuous integration test run reports, parallel test execution for decreased test execution times, extendibility for possibly missing features in the framework and support for all major operating systems.

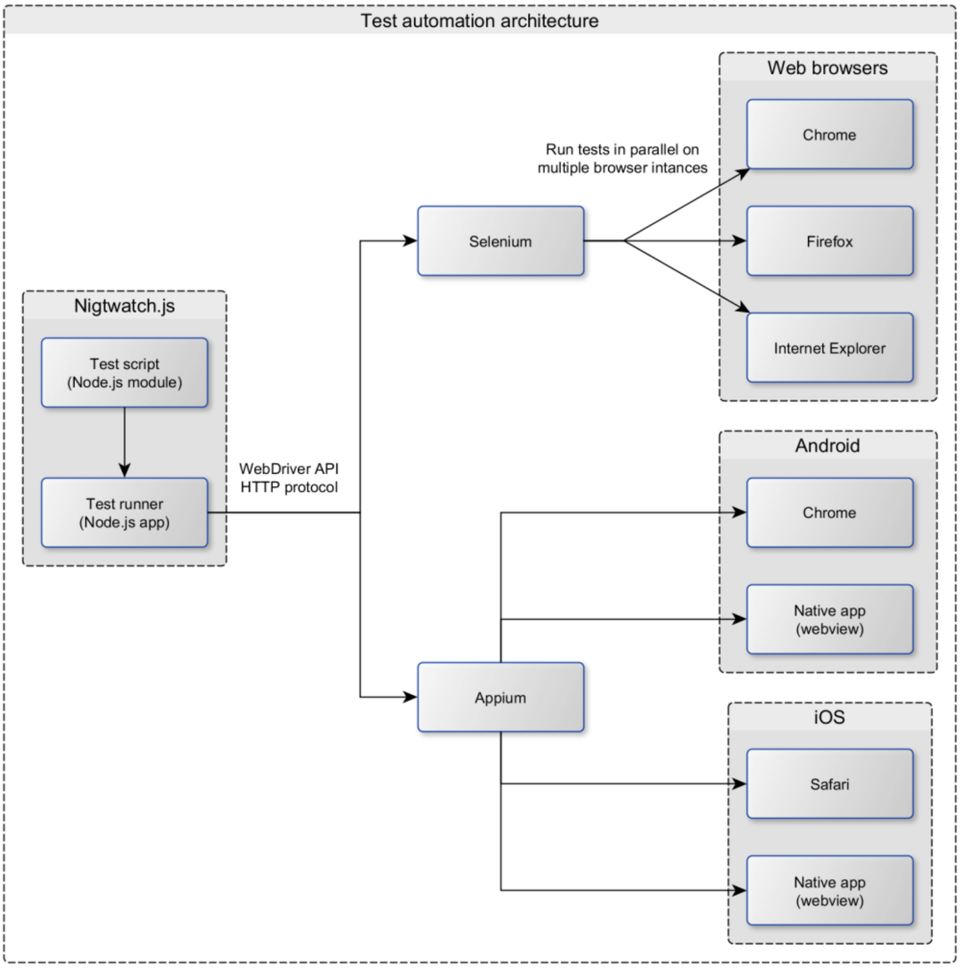


Figure 5: Test automation architecture

Another reason worth mentioning with choosing to go with tools compatible with the WebDriver API is the flexibility it provides. In future projects, some test framework other than Nightwatch.js could be used and it would still be compatible with the existing test automation infrastructure. Also, if needed, Selenium or Appium could be switched to any other WebDriver API standard compatible automation tool.

3.7 Test case design, documentation and implementation conventions

The main motivator for the test case design and test automation was to reduce the number of defects introduced in product releases by increasing newer team members’ knowledge about the projects and the reducing the amount of work required for comprehensive testing. In many of the case projects, the existing feature documentation was very minimal. So, in addition to being the basis for the test automation implementation, the test cases would work as feature documentation for the team working on the projects.

3.7.1 Test cases

An important aspect to be considered in the test case design was the cost and required effort for maintaining the test cases and the automated tests. The more comprehensive the test cases are, the more likely they are to catch regressions, but the more it also costs to maintain them. After introducing written test cases and automated tests any change to an existing feature also means that the related existing test cases and automated tests need to be updated. Also, when new features are added, new test cases and automated tests to be designed and written.

The main guideline for test case design was set to cover all of the basic functionality of all features and the most common exceptions. The automated tests should guarantee that all features work as intended in all of the most common use cases. Drawing the exact line between basic functionality and the most common exceptions and the rarer edge cases would be left to the developer designing the test cases and implementing the automated tests. Not covering every little detail should help keep the required effort for maintaining the tests reasonable.

It was also decided that the test cases would be written as JSDoc comments for the automated tests using Markdown syntax. The automated project build process could then be used to generate HTML documentation pages from the JSDoc comments. It would be the developers who would design test cases and implement the automated tests and the team wanted to keep the test cases and the test automation code as tightly tied up together as possible. This should prevent the test cases from ever being changed without changing the corresponding automation code as well. Also, the test case design would always be covered by the team’s code review process which is applied to all changes in the version control. The simple markdown syntax should also help keep the test cases simple and concise, which should help keep the tests more maintainable. In addition, everyone working on the project would always have the latest test case documentation easily available either through building the project locally or using the documentation generated by the CI tool.

3.7.2 Test automation

The team didn’t have much knowledge of any user interface automation testing tools beforehand. However, a few principles were set before starting the test automation implementation. As with any code base, having a lot of copy-pasted code will lead to a mess and a maintenance nightmare over time. The main principles for writing the automated tests was to organize the code properly and to write as much reusable code as possible.

Writing reusable code would be easier with the Nightwatch.js Page Object API (*github.com*, 2019), which provides an abstraction layer for the test automation code to interact with the UI views. The main principle is that all of the UI views are represented as the page objects which provide functions and handles for the relevant html elements for the automated tests to interact with the UI ("martinfowler.com," 2019). This way the test code can be kept simple and concise and all code related to user interface interaction is in one place for all of the tests to use. Another helpful feature of Nigtwatch.js regarding code reusability is the possibility to extend the core command library. Using these two features properly should a guarantee relatively efficient reusability of code and help avoid writing copy-paste code.

Another decision made beforehand was that the tests would be divided into test suites by feature using the Nightwatch.js test suite support. The division into test suites has several benefits. Having all test cases related to a feature under one suite helps keep the code base organized and easier to understand overall. Instead of always running all of the tests, the tests can be run by suite. When the test base grows bigger, running all of the tests each time could mean tens of minutes of waiting. So when working on changes for a specific feature, over time the team can save a significant amount of waiting by just running the related test suite or suites. Lastly, test suites can be run in parallel, which again means time saved when running the whole test suite.

A known problem in every one of the case projects regarding backend system data was that the backend test environment data isn't static and it can't be reset every time the tests are run. This meant that to prevent the tests from possibly breaking any time, many of the test cases would have rely on mock data and part of implementing the test automation would also be designing the mock data.

4 Method

4.1 Literature review

The Literature section introduces the most relevant testing-related topics regarding this case study: regression testing, user interface testing, test automation and the benefits and the most common pitfalls of user interface test automation. The literature review was conducted using the Google Scholar search engine for scientific papers. Also Aaltodoc, the Aalto University Master's thesis database, was also searched for previous Master's theses related to GUI test automation.

The search was conducted in two different phases. The initial search was performed using a variety of keywords in different combinations. Table 5 shows a categorized list of the keywords used in the searches. The categories weren't used in the searches in any way, but they were helpful for making sure that all different angles and perspectives relevant to this case study were taken into account.

Category	All searches	Type of testing	Descriptive	Technical terms
Search terms	test* autom*	UI user interface test* regression test* functional test*	benefits effectiveness pitfalls cost case study	HTML HTML5 Web DOM Selenium Appium Nightwatch.js

Table 5: Literature review search keywords

Based on the searches, the articles were filtered in three stages based on the relevance to this case study. In the first stage, the articles were selected based on title relevance with very loose criteria. In the next step, articles were selected based on the abstract relevance. Last, the most relevant articles were selected based on the full text of the articles. Table 6 shows the rough number of articles included in each phase of selecting the articles for the literature review.

Phase	Initial search	Title filter	Abstract filter	Selected
Rough number of articles	250	60	30	15

Table 6: Rough number of articles in each selection phase

As the last step of the literature review, the reference lists of the selected articles were searched for relevant articles that were not found with the Google Scholar searches. The articles from the references were selected with the process described above for the articles found with the searches.

4.2 Empirical research

Design science is a widely used practice in IT research. It's a proactive design paradigm that involves solving problems by introducing and evaluating new artifacts in the environment where the problem exists (Hevner and Chatterjee, 2010). Design science is used in this thesis as it suits and is intended for exactly this type of research.

The starting point of research using design science is to recognize and define the problem to be solved and justify its relevance. GUIs are complex and change fast and often and testing them manually is a tedious and time-consuming process. The literature review and case study chapters describe the problems of manual GUI testing more thoroughly and discuss how test automation is expected to solve some of its core issues.

The design artifacts are in the core of design science. The artifacts are the proposed solution to the problem (Hevner and Chatterjee, 2010). The main high level goal of this thesis is to answer the question if the introduction of GUI test automation can be considered useful and beneficial and if it can improve the cost-efficiency of testing. Part of this thesis is also the design and implementation of GUI test automation in the case projects and figuring out how they can be integrated into the current development processes. The case study chapter provides background information and describes the design process and the end results are presented in the results chapter. In terms of design science the two artifacts in this research are the replacement of manual testing with GUI test automation and the actual concrete automated tests designed and implemented as part of the case study.

The last and probably the most important step of design science is the evaluation of the solution and how well it solves the problem (Hevner and Chatterjee, 2010). The evaluation in this research is performed with the evaluation of the designed and implemented GUI test automation. To improve the reliability of the research, the evaluation is split into quantitative and qualitative parts with the goal of them complementing and reinforcing the findings and eliminating the shortcomings of one another. The goal is that the case study provides insight into the usefulness and cost-efficiency of the implemented test automation. The results from the case study are then used to assess their applicability to GUI test automation in general.

4.2.1 Quantitative evaluation: hard data

The quantitative evaluation is based on the hard data collected related to designing, implementing, maintaining and using the test automation. The purpose is to gather relevant data for evaluation of the usefulness and cost-effectiveness of GUI test automation in the case projects. Table 7 presents the data points and their measures used in the evaluation. The time used was split into several different categories for more detailed analysis and evaluation.

The intention was to gather data on the cost-efficiency and usefulness of test automation too, but the case project development teams didn’t yet adopt the use of GUI test automation practices as part of their development processes. Therefore the quantitative evaluation is only limited to the design and implementation phase of the test automation and lacks all data related to its maintenance and use.

Data point	Measure
Total learning and setup time	work days
Total test case design time (by project)	work days
Total implementation time (by project)	work days
Total time used for implementing new tests (by project)	work days
Total time used for test maintenance (by project)	work days
Defects found while implementing the tests (by project)	number of defects

Table 7: Quantitative evaluation data points

4.2.2 Qualitative evaluation: interviews

Because the GUI test automation implemented as a part of this thesis wasn't yet adopted by the scrum teams working on the projects, five of the team members in different roles were interviewed to gather additional data to support this research. The goal of the interviews was to try to assess the team members' knowledge, opinions and attitudes on the benefits, limitations and challenges of GUI test automation and find out the reasons as to why and impediments why it wasn't yet adopted. The interviewees were also asked which steps they think are required for adopting GUI test automation or if it should be adopted altogether.

A total of five team members working in the case project development teams were interviewed. Table 8 shows a summary of the interviewees' roles and experience in test automation and GUI test automation. The interviews were conducted as 15- to 30-minute one-on-one interviews with a loose high-level structure described below. The interviews started with questions about the current state of testing practices in the interviewees' current projects and their knowledge and experiences on the use of test automation to steer their thoughts towards the subject and gain some background knowledge to give some context for their opinions. Next the interviewees were asked about their opinions on the feasibility, benefits, challenges and limitations of test automation and GUI test automation. In the last parts of the interviews, the interviewees were asked if they think GUI test automation should be used at all and what they think are the reasons and impediments that stood in the way of adopting the implemented test automation. Last they were asked which steps should be taken to adopt the use of GUI test automation in the day-to-day processes.

All of the interviews were recorded. The main motivation behind the interviews was to find out the team's opinions on the usefulness, limitations and challenges of test automation and why the test automation wasn't yet adopted and what could be done to improve the situation. The recordings were used to collect a list of points related to these topics for each interview. As the last step, the lists of the individual interviews were compiled into lists on each of the aforementioned topics relevant to this research. The final lists are presented in chapter 6.2.

Interview structure

1. Current state of testing
2. Experience and knowledge in test automation
3. Opinions on suitability, benefits and limitations of test automation
4. Adopting GUI test automation in current projects
    a. Why hasn't it been used more widely yet and should it be adopted at all?
    b. What are the current impediments to using or reasons for not using more GUI test automation?
    c. What concrete steps should be taken to increase use of GUI test automation?

Interviewees

Role	**Experience in the software industry (years)**	**Experience in test automation**	**Experience in GUI test automation**
Scrum Master	5	Low	Low
Scrum Master	3	Low	None
Developer	8	High	High
Developer	3	Low	Low
Developer	1	None	None

Table 8: Interviewees

5 Case project test automation design and implementation

5.1 Test case documentation

None of the case projects had any existing test case documentation. Without test cases there is nothing to automate, so the first step towards test automation was to design and write down the test cases. The complete lack of test case documentation left a lot of flexibility in how the test cases could be documented.

To minimize future maintenance work and lower the chance for the test cases to not be updated with changes to the UI, the main priority for the test case documentation was conciseness, simplicity and clarity. Also, for the same reason the test cases were chosen to be bundled up with the test code and written as JSDoc (“usejsdoc.org,” 2019) documentation comments with Markdown (“daringfireball.net,” 2019) syntax. The intention of this design decision is to eliminate the typical problem of having to update the documentation and the implementation separately and help keep the documentation up to date with the implementation and vice versa.

To follow the principles mentioned above, single test cases should be kept short and simple. A test case shouldn’t try to cover full features, but only test a single use case or other path of using the feature. For example, various paths required for properly testing a form and its input validation should be separated into multiple test cases. A ‘happy path’ test case should use valid inputs and invalid inputs should be tested in another test case.

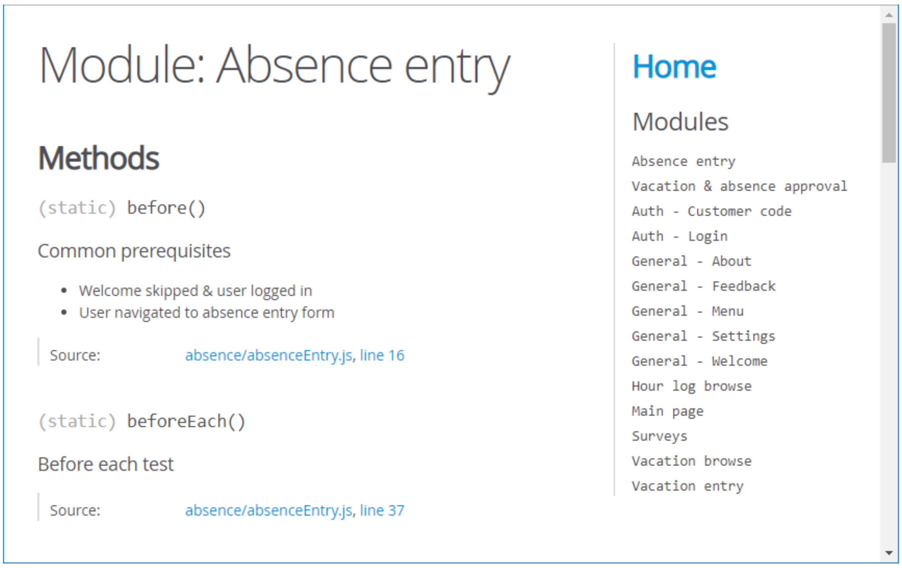
The format for all test case documentation is the same. All of the test cases have a short description of their purpose. Generally, it shouldn’t need to be longer than one sentence or the single test might be too complicated and splitting it should be considered. The test case’s intended test execution is summarized in a table of test steps with possible expected outcomes and test data inputs. Following the principles of conciseness, simplicity and clarity the step descriptions and expected outcomes are mostly described with a few words.

```
/**
 * __Description__
 * Verify the "happy path" for creating a new absence
 *
 * __Test steps__
 *
 * | Step                        | Expected                                       | Data |
 * | --------------------------- | ---------------------------------------------- | ---- |
 * | Select absence type         | Type selected                                  |      |
 * | Select start date           | start date selected                            | 10th day of current month |
 * | Select end date             | end date selected, form save/submit enabled    | 12th day of current month |
 * | Add additional info         |                                                | any text |
 * | Check medical cert checkbox |                                                |      |
 * | Check continuation checkbox |                                                |      |
 * | Click save                  | navigate back to main page, show success toast |      |
 * */
'Absence save success': function(browser) {
```

Figure 6: JSDoc Markdown test case example

The HTML test case documentation is generated with JSDoc from the code documentation comments. The test cases are grouped into test suites. A test suite contains all tests related to one feature or a group of related features. JSDoc generates a separate HTML page for each module. Figure 7 and Figure 8 show examples of module and test case documentation generated with JSDoc. Each test suite is marked as a JSDoc module in the test code so that each test suite gets its own page in the test documentation. Figure 9 shows an example of a test suite declaration.

In the scope of this thesis only local command line documentation generation was implemented. In the future the same command could be run on the CI server and the documentation published as part of the CI pipeline builds.



Module: Absence entry

Methods

(static) before()

Common prerequisites

- Welcome skipped & user logged in
- User navigated to absence entry form

Source: absence/absenceEntry.js, line 16

(static) beforeEach()

Before each test

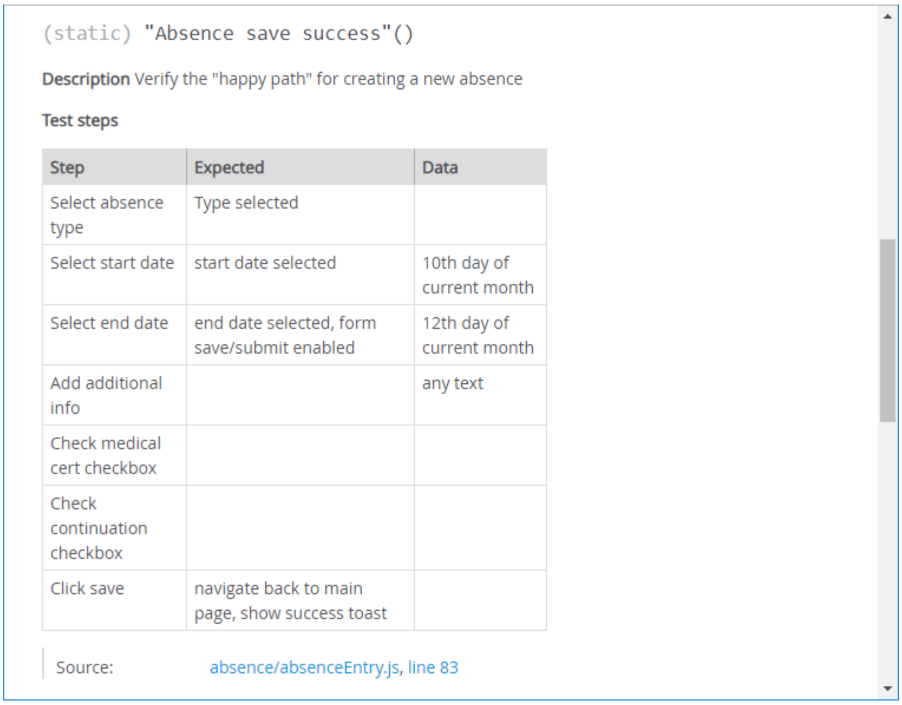
Source: absence/absenceEntry.js, line 37

Home

Modules

Absence entry
Vacation & absence approval
Auth - Customer code
Auth - Login
General - About
General - Feedback
General - Menu
General - Settings
General - Welcome
Hour log browse
Main page
Surveys
Vacation browse
Vacation entry

Figure 7: Generated module HTML documentation



(static) "Absence save success"()

Description Verify the "happy path" for creating a new absence

Test steps

Step	Expected	Data
Select absence type	Type selected	
Select start date	start date selected	10th day of current month
Select end date	end date selected, form save/submit enabled	12th day of current month
Add additional info		any text
Check medical cert checkbox		
Check continuation checkbox		
Click save	navigate back to main page, show success toast	

Source: absence/absenceEntry.js, line 83

Figure 8: Generated test case HTML documentation

5.2 Test design and implementation

As established in chapter 2.4, one of the most common problems with test automation is the time and money required to maintain the existing tests and implement new tests when the UI changes. To tackle the problem, the main focus in the test design and implementation was in code base maintainability, readability and keeping it easy to understand and change. Chapter 3.7.1 describes the principles used in test design and implementation in more detail.

As stated before, the tests were implemented with the Nightwatch.js, which is a JavaScript / Node.js-based web application test automation framework. The framework is compatible with any browser or hybrid mobile test automation server that supports the WebDriver ("w3.org," 2019) standard. In the case projects Selenium ("seleniumhq.org," 2019) was used with the browser tests and Appium ("appium.io," 2019) with the native applications.

The test code level architecture can be divided into the test suites, page objects, custom commands and mock data. The tests are grouped into test suites so that all tests related to a feature are in one test suite. All test logic and test case documentation can be found in the test suites. Page objects function as a representation of the UI views. All interaction with the UI from the tests happen through the page objects. Custom commands are extensions to the base command set provided by the Nightwatch.js test framework. They contain common functionality needed by more than one test suite. Due to constraints of the testing environment many tests also need mock data, which is defined in separate JSON files and their usage can be defined either on a test suite or test case level.

The next four sub-chapters give a more detailed description of how the tests were designed and implemented with some explanations on the design decisions.

5.2.1 Test suites

The test suites gather together all tests related to all the UI views of one feature. In Nightwatch.js the test suites are defined as Node.js modules. The main principle in the implementation of the test suites was that they only contain all non-reusable test code, logic and data. All reusable code and data is grouped into the page objects, custom commands and the mock data files. The intention with this is to keep the tests as maintainable as possible by having the code be simple and readable and avoiding copy-pasting same code to multiple tests or test suites.

absenceEntry.js

```javascript
/**
 * @module Absence entry
 */
module.exports = {
    '@tags': ['absence'],
    /**
     * #### Common prerequisites ####
     * * Welcome skipped & user logged in
     * * User navigated to absence entry form
     */
    before: function(browser) {
        // phantomjs can't clear it's websql so lets help it
        conf.clearPhantomJSCustomerCode(browser);

        absenceEntry = browser.page.absenceEntry();
        absenceEntryPage = absenceEntry.section.absenceEntryPage;

        browser
            .resizeWindow(conf.SCREEN_SIZE.width, conf.SCREEN_SIZE.height)
            .skipToUserLoggedIn(
                conf.CUSTOMER_CODES.VALID,
                conf.USERS.REGULAR,
                'absenceEntry.json'
            );

        browser.page.main().section.mainPage.section
            .absenceCard.click('@newEntryButton');
    },
    /**
     * #### Before each test ####
     */
    beforeEach: function (browser) {
    },
    after: function(browser) {
        browser.end();
    },
```

Figure 9: Test suite example

Each test suite run starts with the application loading and starting as a fresh install, so most test suites need to perform some steps, such as logging in the user, to initialize the app state. To keep the test cases concise and enable them to focus on testing the relevant functionality, the test suites initialize the app to the desired state before starting the execution. This is achieved with the Nightwatch.js framework functionality to run code before and after the execution of test suites and cases. Figure 9 shows an example of this. When the test case prerequisites for the app state are handled by the test suite, the test cases can focus on only testing the relevant functionality.

The test cases use the Nightwatch.js framework core functionality, page objects, common commands and mock data to perform the tests. All UI state checks and interactions happen through the page objects. The page objects hide all complicated actions, such as selecting a date, behind simple functions and provide references to the relevant UI elements. The tests mainly consist of a series of UI interactions, such as a button clicks and modifying form input values, and checks to verify the UI state changes between them. Figure 10 shows the implementation of a typical test case. The goal with how the code is organized and structured is to keep the test case logic easy to follow and the code readable and easy to maintain.

```
'Absence save success': function(browser) {
    //Select absence type
    absenceEntry.selectOption('@absenceTypeSelect', '00016');

    //Insert dates (use current month)
    absenceEntry.selectDate('@startDateInput', 18);
    absenceEntry.selectDate('@endDateInput', 18);

    absenceEntryPage.expect.element('@saveButton').to.not.have.attribute('disabled').after(conf.RENDER_WAIT_TIME);

    //Additional info
    absenceEntryPage.setValue('@additionalInfoInput', 'additional information');

    //Select approver
    absenceEntry.selectOption('@approverSelect', '100000');

    //Check medical certificate
    absenceEntryPage.click('@medicalCertificateCheckbox');
    absenceEntryPage.expect.element('@medicalCertificateCheckbox')
        .to.have.attribute('class').which.contains(conf.CHECKBOX_CHECKED).before(conf.RENDER_WAIT_TIME);

    //Check continuation
    absenceEntryPage.click('@continuationCheckbox');
    absenceEntryPage.expect.element('@continuationCheckbox')
        .to.have.attribute('class').which.contains(conf.CHECKBOX_CHECKED).before(conf.RENDER_WAIT_TIME);

    //Submit
    absenceEntryPage.click('@saveButton');
    absenceEntry.expect.section('@absenceEntryPage').not.to.be.present.before(conf.LOAD_WAIT_TIME);

    const toast = browser.page.toast();
    toast.expectSuccessToastVisible();
    toast.section.toast.waitForElementNotPresent('@toastMessage');
},
```

Figure 10: Test case example

5.2.2 Page objects

The page objects are a feature provided by the Nightwatch.js framework. Their main purpose is to provide a layer of abstraction for the tests to interact with the UI. A single page object represents one UI view and it provides references to the relevant elements in the view and defines commands or actions to interact with the view.

The main goal of the page object commands is to gather all reusable functionality needed by the tests interacting with the view. The purpose is to improve code base maintainability and keep the test cases simple by hiding some of the more complicated UI interaction logic behind simple functions. A good example of this is the date selection command from Figure 11 which involves clicking a date selection input and then selecting a date from the date popup that is only visible after the click. Also, because of how the date selection popup is implemented in the application, the only way to select a date from the test code is injecting and executing JavaScript in the browser that finds and clicks the date element. Selecting a date is conceptually a simple task but it requires some rather complex logic in the test code. When the complex logic is wrapped in the page object custom command it's available as a simple function for all test cases that interact with the view and only needs to be implemented once.

```
let commands = {
    selectOption: function(selectSelector, type) {
        const page = this.section.absenceEntryPage;

        page.expect.element(selectSelector)
            .to.be.present.before(conf.RENDER_WAIT_TIME);

        page.click(selectSelector);
        this.api.pause(conf.DEFER_EVENT_TIME);
        page.click(`option[value='${type}']`);

        page.expect.element(selectSelector)
            .to.have.value.that.equals(type).before(conf.RENDER_WAIT_TIME);
    },
    selectDate: function(input, day) {
        const page = this.section.absenceEntryPage;
        const popup = this.api.page.popup();

        page
            .waitForElementVisible(input)
            .click(input);

        popup.expect.section('@content')
            .to.be.visible.before(conf.RENDER_WAIT_TIME);
        this.api.pause(conf.DEFER_EVENT_TIME);

        //Select date with jquery in the client
        this.api.execute(function(day) {
            $('.ui-popup-active .ui-datebox-grid .ui-btn').filter(function() {
                return $(this).text() === `${day}`;
            }).click();
        }, [day]);

        popup.expect.section('@content')
            .to.not.be.present.after(conf.RENDER_WAIT_TIME);

        this.api.pause(conf.DEFER_EVENT_TIME);
    },
};
```

Figure 11: Page object custom commands

In addition to the custom commands, the page objects also define the elements that are exposed to the tests. The elements are given a name and a CSS selector. The selectors are evaluated when the tests interact with the elements or perform checks on them, so they don't need to be visible or present when the page object is initialized and can be dynamically generated. The main benefits of defining the elements this way is to only define the selectors once and hide the possibly complicated selectors behind simple variables. The changes in the UI can be more easily managed when the view's elements and their selectors are gathered in one place only. UIs tend to change often, and when they change there's a high chance that the DOM structure and the element selectors change as well. Also, the test case code readability is improved when the elements are referenced with names instead of obscure CSS selectors.

```
module.exports = {
    url: `${conf.URL}#absences/add`,
    commands: [ commands ],
    sections: {
        absenceEntryPage: {
            selector: `#addAbsencePage.${conf.PAGE_ACTIVE}`,
            elements: {
                absenceTypeSelect: 'select.AbsenceTypeInput',
                startDateInput: '.StartDateInput',
                endDateInput: '.EndDateInput',
                dateErrorMessage: '.date-error-message-container',
                additionalInfoInput: '.AdditionalInformationInput',
                approverSelect: 'select.ApproverInput',
                medicalCertificateCheckbox: 'label[for="MedicalCertificateInput"]',
                continuationCheckbox: 'label[for="ContinuationInput"]',
                saveButton: '.SubmitButton'
            }
        }
    }
};
```

Figure 12: Page object element structure

5.2.3 Custom commands

The custom commands are a miscellaneous collection of user actions and other reusable functionality needed by the tests. They are implemented as extensions to the Nightwatch.js framework core commands. Their main purpose is to improve the clarity and maintainability of the code base by gathering all code needed in several test suites to one place. Figure 13 shows an example command that is needed in most test suites before the test execution starts. It's a top-level command that utilizes a series of other custom commands to perform all actions required to log the user in after the app is started.

```
skipToUserLoggedIn.js

let conf = require('../e2e/conf');

module.exports.command = function(customerCode, user, mockData = null) {
    this.skipWelcome();
    this.page.customerCode().enterCustomerCode(customerCode);
    this.page.login().expect.section('@loginPage')
        .to.be.visible.before(conf.LOAD_WAIT_TIME);

    if (mockData != null) {
        this.mockData(mockData);
    }

    this.page.login().logIn(user);
    this.page.main().section.mainPage.expect.section('@hourLogCard')
        .to.be.visible.before(conf.LOAD_WAIT_TIME);

    return this;
};
```

Figure 13: Global custom commands

5.2.4 Mock data

Mock data is needed because the backend systems can't be reset before each test execution. Mock data is used to handle any data that can change in the environment. The Nightwatch.js framework doesn't provide any functionality for mocking data, so a custom solution for mocking data had to be implemented.

Mocking data was implemented as one of the custom commands presented above. Since the tests are not executed in the browser or the mobile application directly, but only communicate with them through Selenium/Appium using the WebDriver protocol, it's not possible to mock test data in the test code directly. The mock data implementation injects and executes a snippet of JavaScript along with the test data to the browser window.

The injected JavaScript intercepts all XHR requests using the XHook ("github.com 2," 2019) library and returns mock responses for the ones that match the responses for the HTTP Methods and paths defined in the mock data file. So, the app itself doesn't see any difference in requests that are mocked and the ones that aren't. The test suite or test case defines the mock data to be used, but it doesn't have any special handling for the mock data either. All elements are used and interactions function the same way regardless of the use of mock data.

```
module.exports.command = function(dataFile) {
    let data = fs.readFileSync(conf.MOCK_DATA.DIRECTORY + dataFile, 'utf8');

    let initScript = fs.readFileSync(INJECT_INIT_SCRIPT_FILENAME, 'utf8');
    let xhookLib = fs.readFileSync(conf.MOCK_DATA.DIRECTORY + conf.MOCK_DATA.LIB_XHOOK_PATH, 'utf8');

    initScript = initScript.replace('{INJECT_API_URL}', conf.API_URL);
    initScript = initScript.replace( searchValue: '{INJECT_MOCK_RESPONSE_DELAY}', conf.MOCK_DATA.RESPONSE_DELAY);
    initScript = initScript.replace( searchValue: '{INJECT_MOCK_DATA}', data);
    initScript = initScript.replace( searchValue: '{DEBUG}', conf.MOCK_DATA.DEBUG);

    this.execute(xhookLib);
    this.pause(200);
    this.execute(initScript);
    this.pause(200);

    return this;
};
```

Figure 14: Mock data custom Nightwatch.js command

```
require(['xhook'], function(xhook) {
    xhook.before( nodes: function(request, callback) {
        let apiPath = request.url.split(API_URL)[1].split( separator: '?')[0].split( separator: '/');
        let mockData = MOCK_DATA[request.method];

        if (mockData != null) {
            for (let i in apiPath) {

                if (typeof mockData[apiPath[i]] === 'undefined') {
                    if (typeof mockData['*'] !== 'undefined') {
                        mockData = mockData['*'];
                    } else {
                        mockData = null;
                        break;
                    }
                } else {
                    mockData = mockData[apiPath[i]];
                }
            }
        }

        if (mockData != null) {
            setTimeout( handler: function() {
                callback({
                    status: mockData.status,
                    text: JSON.stringify(mockData.data),
                    headers: {
                        "content-type": "application/json;charset=UTF-8"
                    }
                });
            }, MOCK_RESPONSE_DELAY_MS);
        } else {
            console.log('no mock data');
            callback();
        }
    });
```

Figure 15: Mock data inject script core logic

Figure 16: Example mock data definition

5.3 Problems encountered while implementing the test automation

This chapter presents the problems encountered in the test automation implementation. They can be categorized into problems related to GUI test automation in general, the Nightwatch.js test framework and the tools used for running the tests. Table 9 summarizes the problems under each of the categories.

GUI test automation	**Test framework (Nightwatch.js)**	**Testing tools (Selenium, Appium)**
Designing and implementing test automation is a complex and time consuming task	The wait commands for UI state transitions didn't always work as expected	Test execution is somewhat unreliable in some cases
Test data and environments caused more problems than expected	Some of the Nightwatch.js core functionalities had bugs	Browser updates on the machines running the tests can break the tests
	Missing nice-to-have features: touch events and clear input	Connection from Appium to the emulator lost between test suites
	With the way the framework is designed, the test code and the JavaScript scopes get very complicated sometimes	

Table 9: Problems encountered in test automation implementation

5.3.1 GUI test automation

Two of the encountered problems can be generalized under the category of GUI test automation. Designing and implementing the tests took considerably more time than expected due to the complexity in the test case details and learning the tools and their limitations and quirks. Even though the guidelines for the scope of testing were set before starting the test case designs, designing the tests and their exact steps and verification criteria took a considerable amount of time. Chapter 6.1 presents the detailed numbers related to test automation design and implementation.

The capabilities of controlling the test environments were known to be limited, but figuring out the means for working with mock data was surprisingly complex and time-consuming. Choosing which data should be mocked and designing the mock data added an extra layer of work for each test case. Also, the test framework didn't provide any means for mocking data, so a custom solution described in chapter 5.2.4 was implemented.

5.3.2 The test framework

The second category of identified problems in the test case implementation are the issues and limitations encountered while working with the selected test framework. One of the most frequently encountered issues with implementing the tests was that in certain situations some of the actions or state checks on the UI elements can't find the element even though they are clearly visible on the screen and present in the page DOM tree. This happened regardless of the principle of always verifying the visibility or presence of the element before performing further actions or checks on them. For example, a triggered click event on a button might never be registered even though the button element's visibility and existence was checked with the previous line of executed code. Also, in some situations even the element visibility and presence checks failed without any obvious reason. The root cause for the problem was never found regardless of countless hours put into struggling with it. In some cases the cause could be that the element JavaScript click listeners weren't yet initialized even though it's visible. The failed visibility checks on visible elements can't be explained with this though, so it's likely that there's some issues in the framework, its WebDriver protocol communication with Selenium or Selenium returning invalid responses to the framework. The workaround for the problem was to add constant timed waits in the test script execution just before the problematic commands. This problem was one of the most time-consuming to deal with as it was usually hard to tell whether the problem was an actual fault in the test code logic or another instance of the issue.

In addition to the above-mentioned problem the Nightwatch.js framework was missing some core functionality and some functionalities didn't work as expected. For example, the framework didn't have commands for clearing an input field and the input field set value command appended the value to the input field instead of replacing the value. Custom commands were implemented to get the desired functionality needed with the test cases. Also, the framework didn't support touch events, so anything involving swipes and other gestures had to be scoped out of the test cases.

Another issue worth mentioning working with the Nightwatch.js framework is the complexity of its code execution order and how its command queue works. The framework executes all test case code and the commands are added to the command queue before the test execution starts. When the test execution starts, the commands are executed one by one. Working with test steps that require data from one or more of the previous steps involve setting up callbacks and the test developer has to be very careful with which variables and commands are available in which scopes. This adds an extra layer of complexity to learning the framework and designing the test cases and increases the learning curve of writing tests with the framework.

5.3.3 Testing tools

The test runner tools Selenium and Appium seemed to be somewhat unreliable at least when used with the Nightwatch.js framework. Some of the tests runs failed for no evident reason. The issue didn't seem to be tied to any certain test cases, but the probability for it to occur seemed to increase with the length of the test case. The only workaround for the issue was to add a few retries for the test cases and mark them as failed if none of the tries pass.

In the timespan of the few months of time used for the test automation implementation, browser updates broke the tests several times. Selenium requires a WebDriver for each browser and usually the WebDriver needs to be updated with the browser. All modern browsers update frequently and automatically, so the WebDrivers easily get out of sync with the browser versions on the developer's computer. In none of the cases when an outdated WebDriver version broke a test case, did an error message point to that direction. Especially the first times when this happened, connecting the previously working, now broken, test case to this issue took a lot of extra time.

The last observed issue when testing the native builds was that Appium's connection to the Android emulator was lost and timed out frequently in the test execution. This didn't break the tests, but slowed their execution time in these cases to over ten-fold as the automatic reconnection worked reliably, but the default connection timeout was long and wasn't configurable. The reason for the issue was never found.

5.4 Test maintenance and execution plan

Currently regression testing is only carried out as part of the version release process. The plan is to integrate test automation into the software development process. Figure 17 describes the updated software development process. Otherwise it's the same process as previously but updating old tests and creating new ones and then running the automated tests is part of the task development process now. Running test automation is also added as a step to the QA review process after manual testing and code review. If any issues are found by the test automation they will be fixed as a part of the process.

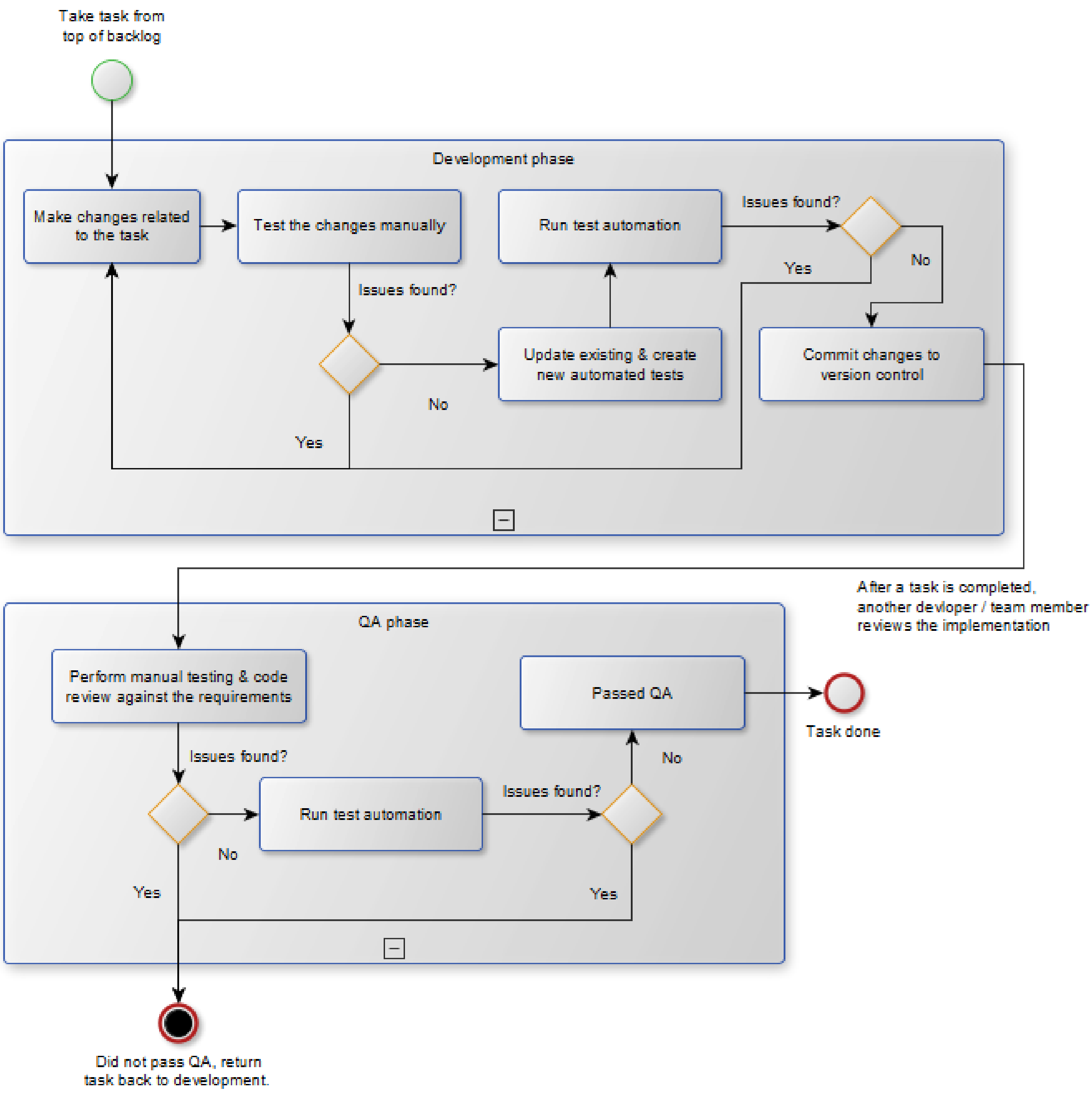


Figure 17: Updated software development process

The two main goals with the process design was to ensure that the test automation is always kept up to date and that the development-testing feedback loop is kept as fast and short as possible. As established in chapter 2.4, a typical challenge when introducing test automation is that the tests aren't updated and new ones aren't created when the GUI is changed. When test automation isn't updated with the changes, the tests are highly likely to break and become obsolete. Also updating and creating the tests right away should require less work in total as the requirements and other details of the implementation are fresh in the mind of the developer. If the test automation is updated later out of sync with the development process, the same requirements and other details need to be learned again.

The short feedback loop of the planned processes should help with maximizing the benefits of test automation. When the issues are found and fixed as part of the development process, the quality delivered by the process should be higher, since not as many issues should get through it. Also, even though the process of completing a development task is longer and involves more steps, on long-term running the tests as part of the development process should allow for faster delivery of changes and new features, because the team needs to spend less time fixing issues introduced in earlier changes.

6 Results: Quantitative and qualitative data

This chapter presents the results to support the qualitative and quantitative assessment of the usefulness of the test cases and to provide insight into why the teams didn't adopt the use of the implemented test automation yet. The main purpose of this data is to help answer the research questions. Chapter 6.1 presents the data related to implementing the test automation. The initial plan was to also collect data on the cost-efficiency and usefulness of the test automation in use such as the maintenance costs, the saved time in manual testing and number of issues caught by the test. However, as the scrum teams didn't yet adopt the new practice, the results from the interviews presented in chapter 6.2 provide insight on the reasons why the use of test automation wasn't yet adopted even though that was the intention.

6.1 Hard data

This chapter presents the data collected on time usage and other metrics about the design and implementation of the test automation. The data is presented in the four tables below. Table 10 shows the number of test suites, test cases and assertions for each case project. These numbers should give some context for comparing the amount of work done for implementing GUI test automation for each case project presented in the table and Table 12.

Project	Test suites	Test cases	Assertions
Project 1	10	53	726
Project 2	15	70	725
Project 3	8	28	269
Total	33	151	1720

Table 10: Test statistics by case project

Table 11 presents the time used for all planning and preparation activities that aren't related to any of the case projects. These numbers are presented to give a better picture of the total costs of designing, implementing and adopting test automation.

Preparation task	Days used
Test case documentation plan	1
Install and configure tools	1
Learning the basics for the tools	4
Maintenance and integration plan	0,5
Total	6,5

Table 11: Work days used for planning and preparation tasks

Table 12 shows the number days used for designing and implementing the test cases for each project. The purpose of these numbers is to help assess the cost-efficiency of the test automation. Table 13 shows the number of test cases designed and implemented per work day calculated from Table 10 and Table 12 data.

Project	Test case design (days)	Test case implementation (days)	Total (days)
Project 1	3	19	22
Project 2	5	11	16
Project 3	1,5	3	4,5
Total	9,5	33	42,5

Table 12: Work days used for designing and implementing the test automation

Project	Test case design (cases / day)	Test case implementation (cases / day)	Total (cases / day)
Project 1	17,7	2,8	2,4
Project 2	14,0	6,4	4,4
Project 3	18,7	9,3	6,2

Table 13: Test cases per day ratio

Table 14 shows the number of bugs found in the implementation of the automated tests. The number of issues found in the implementation phase is one of the metrics for assessing the usefulness of the tests and their impact on the product quality.

Project	Bugs found in test implementation
Project 1	4
Project 2	5
Project 3	0

Table 14: Bugs found while implementing test automation

6.2 Interviews

The case project development team member interviews were conducted to gather more data on the usefulness of UI test automation and why the implemented test automation wasn't yet adopted. The main focus of the interviews was to gain insight on the development team's previous experiences with both GUI and other test automation, their opinions on the subject, why they think test automation wasn't yet adopted and what concrete steps are needed to adopt its use as part of the team's day-to-day development practices. To take different points of view into account, the interviewees were selected so that team members in different roles, including Scrum Masters and developers, with varying levels of experience were included.

Only three of the five interviewees had previous experience in GUI test automation. So some of the views were based on the interviewees' previous experiences with it, some on generalized experiences with other types of test automation and some on what they had read and heard about it.

6.2.1 Benefits of GUI test automation

The interviewees were asked about their opinions on the possible benefits of GUI test automation. Table 15 summarizes the benefits mentioned by the interviewees.

Benefits of GUI test automation
1. Reduces the time required for manual testing
2. Can help manual testing to be focused on more essential and complex test cases
3. Can speed up software development
4. Can improve delivered quality
5. The automated test cases can work as documentation for the features they are testing
6. Test automation makes it safer to make changes
7. Automated tests eliminate the chance for human mistakes in test execution
8. GUI test automation is the only way to perform automated end-to-end testing for applications with GUIs

Table 15: Interviews - Benefits of GUI test automation

All interviewees mentioned reduced time required for manual testing as a benefit of GUI test automation. It is also strongly related to the second point in Table 15, which is that GUI test automation can help manual testing to be focused on more essential and complex test cases when the basic use cases are covered by automation. The reasoning is that the test automation handles all basic use cases and eliminates the need to test them manually. The time freed up by automation can be used on manually testing complex use cases not feasible to automate.

According to the interviewees, GUI test automation can also speed up software development and improve delivered quality. The thinking behind these views is that when the tests are run more frequently and preferably as a part of the software development process, the problems and bugs are potentially caught and fixed earlier. While verifying changes with test automation and fixing the problems caught by them might increase the cost of making changes, the idea is that that time is saved through the reduced number of issues that need to be fixed later when they're found. Ideally, this lets the team focus more on developing new features instead of fixing issues with previously implemented ones. Also, the argument for improved quality is that when more issues are caught and fixed earlier, less issues should end up in releases and therefore the end-users should encounter fewer issues.

Two of the interviewees also mentioned that test automation can also serve as documentation for the features they are testing and make it safer to make changes. Both of these are helpful especially for new developers in the project, but should be useful for all development team members especially in longer and bigger projects. When a projects grows past a certain point in size or goes on for a long time, eventually it is just too big for everyone to know or remember the details about every feature. When this happens, test automation can help the developers gain knowledge on how the feature should work and feel safer and more confident making changes as the tests guarantees that the changes won't break the functionality covered by them.

Another point from the interviewees was that automated tests aren't prone to errors like manual test runs. Especially when testing is carried out often, executing the same tests over and over again can be considered a tedious and boring task. With manual testing there's always the chance that the developer decides to deviate from test case steps for the aforementioned or any other reason. Also even the most rigorous tester can make a mistake in test execution.

Lastly, one of the interviewees mentioned that GUI test automation is the only way to perform automated end-to-end testing for the whole system for applications that have GUIs. It might seem self-evident, but it's a point worth bringing up. All other methods for test automation such as code-level unit testing or API integration testing only involve testing parts of the system in isolation or testing how one or more of the parts work together. The benefit of end-to-end testing is that it verifies that all parts of the whole system work together as intended.

6.2.2 Limitations and challenges of GUI test automation

This chapter presents the interviewees responses on the limitations and challenges of GUI test automation. Table 16 summarizes the list of limitations and challenges compiled based on the interviews.

Limitations and challenges of GUI test automation
1. Designing, implementing and maintaining GUI test automation is challenging and time consuming
2. Requires deep domain knowledge of the SUT and the technologies used in its implementation
3. Test case selection for automation is hard and time consuming
4. Test automation tools are complex, lacking, have a high learning curve and require a considerable amount of time to learn
5. Challenging to introduce and adopt new processes
6. Limited capabilities of verifying certain common important NFRs of GUIs like usability, animations, layout feasibility or conformance to the design guidelines
7. Only catches the problems that the tests are designed to catch
8. GUIs are tend to change often, so the tests need to be changed often as well
9. The tests break easily

Table 16: Interviews - Limitations and challenges of GUI test automation

The first four points in Table 16 are all related to each other. All interviewees agreed that designing, implementing and maintaining GUI test automation is challenging and time-consuming. The main reason for this given by most of the interviewees is that UIs are much more complex than APIs or smaller sections of code typically covered by other types of test automation. UIs typically have too many different states and states too complex to cover them all with test automation, so the test cases need to be selected and designed carefully for them to cover all relevant states and transitions with a minimal number of tests.

Test case design and selection is tightly tied to the cost-efficiency and usefulness of the test automation. A minimal set of tests is fast and cheap to implement, but the downside is that it won't cover the relevant functionality and catch as many issues as a wider set of tests would catch. Working with GUI test automation requires deep domain knowledge of the SUT and the technologies used in its implementation, so that the test automation covers the relevant functionality with an optimal number of test cases and leaves the most complex cases for manual tests.

The automation tools bring another layer of complexity to the implementation of test automation. All interviewees with experience in GUI test automation agreed that the tools they had experience in are complex and lacking. They have a high learning curve, lack features or have other limitations. This significantly increases the amount of time required for any team to adopt use of GUI test automation. One of the interviewees speculated that GUI test automation tools aren't yet as mature and functional as typical tools used for unit testing and other test automation, because automated GUI testing isn't as widely used a practice in the industry as other types of test automation.

Point 5 from Table 16 was brought up by both of the interviewed Scrum Masters. According to them, it's always challenging to introduce new shared team processes and practices. People naturally resist any change and changing their ways of working is always a push and takes effort and energy. It simply isn't enough to decide that the team should adopt a new practice, but a clear path and steps to get there need to be agreed on and it needs to be someone's responsibility to see to it that the change really takes place. This isn't specifically related to GUI test automation, but applies to it as well.

Points 6 and 7 in Table 16 are both related to the limitations of test automation. Test automation has limited capabilities of verifying certain common important NFRs of GUIs like usability, animations, layout feasibility or conformance to the design guidelines. This sets restrictions on what can be covered by test automation and what needs to be tested manually. Another restriction pointed out by two of the interviewees was that a human eye can always catch and notice problems that aren't part of the actual test case even without deviating from the test case steps in any way. An automated test will only catch the problems that it is designed to catch.

The next points, 8 and 9, were brought up by two of the interviewees. GUIs tend to change often, so the tests need to be changed often as well. Also, GUIs are complex and have many intertwined functionalities and dependencies. A quote from one of the interviews explains it well: "*A change in one part of the GUI code is more likely to have unforeseen consequences than a piece of software with a more defined functionality like a REST API.*" Another point brought by another interviewee was that in web and mobile development, the browsers and operating systems are quite fragmented and updated frequently. Any minor change in those might break the tests. These reasons both negatively contribute to the maintenance costs and the cost-efficiency of GUI test automation.

6.2.3 Opinions and claims about GUI test automation

This chapter presents miscellaneous claims and opinions on GUI test automation from the interviewees that don't directly fall under the category of benefits, limitations or challenges. All of the interviewees believed that even though GUI test automation requires a high upfront investment in time, it can be beneficial and cost-efficient and save time if planned and carried out correctly. None of the interviewees believed that test automation could replace manual testing altogether. However, their common belief was that over time the time saved in manual testing will exceed the initial investment and the time required for implementing new tests and maintaining the existing ones. A couple of the interviewees also mentioned that test automation is especially useful for regression testing. Automated test can be carried out more often compared to manual testing. The saved time in the execution of the tests enables running the tests more often.

All interviewees thought that GUI test automation is more useful in bigger projects with long lifespans and several iterations of development. The interviewees' general opinion seemed to be that the bigger, longer and more long-lived the project is, the more useful test automation is. Also, most of the interviewees said that test automation probably shouldn't be used at all in projects with short lifespans and only one development iteration.

There were contradicting views between interviewees on whether test automation should be introduced in a project from the beginning. The argument for using test automation from the start of the project is that it can be time-consuming and hard or impossible to introduce test automation afterwards in a project. The other view was that GUIs tend to change too much in the beginning of the project, so test automation should be introduced after the GUI has stabilized and doesn't change too much anymore. The ones with this view pointed out that testability still needs to be taken into account from the beginning of the project, or introducing test automation might be problematic afterwards.

6.2.4 Impediments for adopting GUI test automation in the case projects

The last part of the interviews focused on the impediments for adopting the GUI test automation implemented as a part of this thesis in the case project development teams. All but one of the interviewees thought that using GUI test automation could be beneficial and the team should try it, so the reasons for the case project development teams not adopting the use of test automation yet had to be somewhere else. Table 17 lists all impediments for the case projects teams adopting GUI test automation.

Impediments for adopting GUI test automation in case projects
1. No existing processes for use of test automation
2. Adoption of test automation is no one's responsibility
3. Case project organization structure hinders the introduction of test automation
4. Introducing test automation in ongoing projects requires too big upfront investments
5. Testing and test automation is boring
6. Very little experience in test automation in the scrum teams
7. Lacking test environments

Table 17: Interviews - Impediments for adopting GUI test automation in case projects

The case project development teams had no existing processes for using test automation or GUI test automation. Introducing new processes is always challenging. According to one of the interviewed Scrum Masters there are many open questions related to the test automation processes like “When and by whom are the tests designed and written?”, “When and by whom are the tests updated?” and “When and by whom are the tests executed and how are the results used and reported?”. The other Scrum Master also pointed out that another factor making the adoption a more complex process is the Scrum teams’ common goal of having unified processes. Whenever new processes are introduced, they should be discussed among all of the mobile development teams. One of the interviewees described their unclarity on how test automation should be used and how the test cases should be selected using test automation with these words: “*At this point everything related to it feels like a cloud of dust to me. We should first clarify all of this for everyone, so that we’re on the same page and start working towards a common goal.*”

In addition most of the interviewees mentioned that one of the main problems with adopting test automation is that it’s no one’s responsibility to advance it. It’s going forward with its own momentum and it’s up to the activity of individual scrum team members to push it forward. Based on the interviews, the problem seems to be that they don’t have much power and many seem to have lacking motivation to do so. There’s very little experience and existing expertise in test automation and automation tools in the development teams. The progress is very slow because of the aforementioned reasons. It’s hard to find the drive for pushing forward a complicated process when you don’t have strong expertise, opinions or much previous experience with the subject. On top of the complexities of adopting new processes and the lack of responsibility, no one has allocated time to work on pushing the adoption of test automation forward in the development teams.

Introducing test automation in ongoing projects requires a huge upfront investment in selecting and learning the testing tools and designing and implementing the tests. One of the interviewees said the following: “*Test automation can’t be introduced little by little as a side activity with other* *development tasks like code refactoring and other minor improvements, but adopting it requires a huge amount of time to set up and implement test automation for existing features.*" With this, the interviewee implies that adopting test automation is such a big task that it requires specifically allocated time for it. This impediment doesn't apply to the projects for which the automated tests were implemented as a part of this thesis, but they represent only a fraction of the products that the scrum teams maintain and actively develop.

Another point brought up by four of the interviewees including the two Scrum Masters is that the project organization structure hinders the introduction of test automation. The organization structure was described in chapter 3.1. The two Scrum Masters said that the backend system OUs have a strong say in project priorities and responsibilities. Both of the Scrum Masters also stated that features, changes and fixes are prioritized over using time for implementing and maintaining test automation even though the use of test automation has been discussed with all project stakeholders and everyone agrees it should be part of the processes. A quote from one of them describes the situation well in one of the projects: "*The use of test automation has been discussed, but no one seems to be willing to make a decision to allocate the needed time and resources for it.*" The interviewees' answers didn't clearly state whether this has been an implicit decision in all other projects too or if it's just a side-effect of the development team prioritizing work as the result of a pressure to get new features out. Anyway, it was clear based on the answers that there's no budget for introducing or using test automation in any of the projects.

One of the interviewees also had a very different view on the usefulness of test automation in the current projects and why the use of test automation hadn't been prioritized higher. They claimed that the current level of testing and delivered quality is sufficient and no test automation is needed and that these types of business-to-business (b2b) applications don't need similar kinds of quality assurance processes as consumer products where the success heavily relies on end-user satisfaction. The users of these types of b2b applications don't have the possibility to switch to using another application if the quality isn't top notch and the product completely bug-free. The interviewee implied that it's a business decision and that improved quality probably wouldn't improve sales of the products and the return on investment would only decrease with the introduction of test automation.

Another impediment mentioned by two of the developers is that testing and test automation is seen as a boring task by most developers. It's more exiting to implement new features instead of ensuring the correctness of the existing functionality. One of the interviewees gave a concrete example from one of the projects he's working on where all developers don't update or run the existing unit tests in one of the projects and they end up breaking often and have to be fixed later on.

The last point brought up by one of the interviewees is that the test environments are lacking. There are no isolated end-to-end test environments that could be reset either periodically or on command for any of the products. This same issue was encountered while implementing test automation for the case projects too (see chapter 5.3). Test environment data is mostly managed manually. This imposes a problem for test case design as any non-static data might change in the backend systems at any time. Because of this, mock data needs to be used and designed with test cases. Mock data design is time-consuming and requires deep domain knowledge on the SUT.

6.2.5 Steps to advance adoption of test automation

When asked for concrete steps to advance the adoption of test automation, the interviewees came up with a total of two suggestions. All but one of the interviewees suggested that the best way to make progress is to set clear responsibilities for one or more of the team members to push the adoption of test automation forward and allocate enough time for the team to do so.

The other suggestion from two of the interviewees was to wait for project downtime to get the test automation up-to-date with the features and after that use the test automation and create new and update old tests as part of the development processes. They didn’t think it’d be possible to get time allocated specifically for test automation before there was nothing else to do in the projects.

7 Conclusions & discussion

The results point to similar findings as previous research and literature. The collected data confirms that the design and implementation of test automation is an expensive and time-consuming task. Also, the issues identified in the implementation process are similar to the findings in previous research.

Chapters 7.1 and 7.2 discuss the results of the literature review, the test automation implementation and the interviews from the two research questions' ("Can use of GUI test automation be considered beneficial and cost-efficient and which are the main factors affecting this?" and "What are the typical impediments to and challenges of adopting and using GUI test automation?") point of view. Chapter 7.3 discusses the threats to the validity of this research and the need for future work.

7.1 Usefulness and cost-efficiency of GUI test automation

One of the main goals for the GUI test automation in the case projects was to design and implement it with long-term cost-efficiency in mind. As established in chapter 2.4.2, fast and easy implementation of test automation leads to poor maintainability and a higher chance of the adoption of test automation to fail (Kasurinen et al., 2010). Therefore, to increase the chances of the test automation being useful and cost-efficient in the long-term, the main principles while working on the automation were reusability and maintainability. Concrete examples of those principles in use are the test case documentation practices and the test setup architecture and organization of code presented in chapters 5.1 and 5.2.

The test cases use a concise and lightweight documentation format with focus on only the most important details related to the test case. Also, the documentation resides in the test suite code files right next to the test case implementation logic. This should help with always keeping the documentation up to date with the test case implementation. The main goal with these two design decisions is to make it as easy and low-effort a task as possible to keep the test case documentation and implementations up to date.

The test case setup code level architecture was also designed with reusability and maintainability as two of the main goals. The page objects create a layer of abstraction between the test cases and the GUI and they help keep all of the complicated GUI interaction logic in the page objects, so that all of it only needs to be written once and can be used from all test suites and cases. Another example of this are the custom commands and utility functions created. With these two design decisions, the test cases are as simple and concise as possible and only hold logic crucial to the test case execution, while and all complicated interactions are handled by the page objects and the custom commands where the code could be easily reused.

As Table 12 shows, the hard data collected on the implementation of the test cases confirms that designing, setting up and implementing test automation requires a significant amount of work. This was expected, because almost all previous research agrees on the high upfront investments required for introducing test automation as stated in the literature review chapter 2.4.2. Also the principles of maintainability and reusability in the test case setup design reflected on the total implementation effort by increasing it as expected.

One interesting finding from the numbers presented in Table 13 is that even though the work days for setting up and learning the basics of the testing tools weren't counted in project 1's implementation time, the number of test cases implemented per day increased by 129% from case project 1 to case project 2 and increased again by 45% from case project 2 to case project 3. The test case implementation speed was only 2.8 test cases per work day for case project 1 when the same number for project 3 was 9.3 test cases per work day. The test cases for each project are roughly the same size and have the same complexity on average. These numbers can be explained with two factors. First, as also supported by the findings of the literature review in chapter 2.4.2, the testing tools have a high learning curve and designing and implementing test automation is a complex task. The efficiency of the work increases with the gained knowledge and skills required to use the tools efficiently. Chapter 5.3 describes the problems encountered while working with the test automation. Many of the issues encountered were easier to work around and avoid after running into them a couple of times. Another factor contributing to the decreased work required per test case was the reusability of the code. Almost all of the custom commands, including the mock data implementation was reusable in the next projects once implemented while working with project 1 and 2. Also, the since the GUIs consist of very similar basic building blocks, certain code structures and designs were repeatable with similar kinds of UI elements between the projects even though the code couldn't be used as is.

Unfortunately the data on the cost-efficiency and usefulness is close to non-existent as the test automation wasn't yet adopted by the development teams. The only small sign of the usefulness of the test cases were the few bugs found in the test automation implementation.

Apart from high implementation effort and the signs of decreased effort required for working with the test automation after learning the tools and amassing a reusable codebase, not much can be concluded from the data related to the usefulness and cost-efficiency of the test automation in the case projects. Especially for GUIs, test case maintenance forms a significant share of the total costs of adopting and using test automation (Berner et al., 2005). Any credible estimate of the cost-efficiency and usefulness of the test automation has to include data on the realized benefits and costs of maintenance. Therefore, it is evident that the possible benefits and cost-efficiency aren't in any way visible in the results, but would require data on the long-term use and maintenance of the test automation.

7.2 Challenges and impediments of adopting and using GUI test automation

All of the challenges, limitations and impediments of test automation that were encountered in the implementation of the tests and that were collected from the interviews were also identified in the literature review. The main challenges in the implementation of the test automation were the complexity of the tools, test environments and test data and the limitations and restrictions of the test framework and testing tools. All of them increased the total effort of setting up and implementing the test automation.

Kasurinen et al. (Kasurinen et al., 2010) reported that many companies end up developing their own test automation tools or extending existing ones. The Nightwatch.js framework was also lacking in features and had other issues reported in chapter 5.3. Extensions to the framework, such as missing basic functionality and the ability to mock test data, had to be implemented to effectively use it for GUI test automation in the case projects.

Another major challenge in the implementation was the lack of test environments where test data could not be reset on command. Because of this, mock data had to be used to test any functionality that used any dynamic data from the backend system. If this aspect of testability had been taken into consideration from the beginning of the projects, the additional step of designing mock data for each test case wouldn't have been necessary. Similar issues with test environments and test data were also identified by Liebel et al (Liebel et al., 2013). The implications of the lack of testability was also identified as a common challenge with test automation in the literature review chapter 2.4.2. In addition to added effort, the use of mock data decreases the value of the tests as end-to-end tests and reduces them to GUI-only tests. This decreases the potential usefulness of the tests in the future as GUI test automation as parts of the whole system are cut out of the scope of the tests. Typically GUI test automation finds issues in the underlying systems in addition to issue in the GUI itself (Brooks et al., 2009).

The main goal of the interviews was to provide insight into the development teams' attitudes and opinions on GUI test automation and what they think are the main impediments and challenges of adopting test automation in the case projects. The first finding of the interviews was that all but one of the interviewees thought that GUI test automation could be beneficial in the case projects and that the teams should try it at least. Therefore the explanation wasn't just as simple that the teams didn't want to adopt the use of test automation into their development processes.

The main impediments listed by the interviewees were the lack of responsibility, lack of allocated resources, organizational barriers and lack of test automation skills in the development team. One of the main issues seemed to be that no one had been assigned the active role and responsibility and given the resources and time to push the adoption the use of GUI test automation forward. Adopting new processes in any development team always requires planning and effort and when there is no one to push it forward, everything just stands still and no changes happen or they happen very slowly. It's especially true for changes of this scale that require the whole development team to learn something completely new and integrate it into their everyday development processes. This is a typical issue when introducing test automation to a team with no previous experience of it and it was also confirmed by the findings of the literature review. The adaptation of the team's processes for use of test automation requires time and effort (Rafi et al., 2012; Wiklund et al., 2017). The team's lacking skills in test automation also make adapting the processes harder. Without previous experience with the subject, it's hard to know where to even start and many of the details related to the subject can be unclear. It's hard to push new processes forward when it's unclear where to start and what to do.

The other two impediments – the lack of allocated resources and organizational barriers – are both related to each other. The interviewees explained that the mobile development teams working on the case projects don't have the power to decide over the prioritization of the work in the projects. All of the mobile applications of the mobile team are developed in collaboration with the backend system organization unit (BSOU). The BSOU has the ownership of the backend system and the mobile applications are seen as extensions to it, so the BSOU typically has the final say in the prioritization of work in the project. In addition, the BSOUs are typically responsible for the acceptance testing of the mobile application since they have better domain knowledge on the business area of the application. These two facts combined, development work, fixing issues and other changes are prioritized over testing and working on test automation. Similar organizational issues were identified in the literature review as well. Prioritization of work with more immediate benefits and results such as feature development over work that could have long-term benefits is a common issue in software development projects (Wiklund et al., 2017). Haugset and Hanssen also discovered similar issues in their case study on automated acceptance testing: "*One customer felt that it was easy to put off writing them because they were complex beasts*" (Haugset and Hanssen, 2008).

When combining the challenges and limitations identified in the implementation of the test automation and the results interviews, the list is identical to the ones presented in the literature review with a few exceptions. The three exceptions are unrealistic expectations of what can be achieved with test automation and incorrect testing strategy and issues with maintenance effort and costs. The last two could still happen when the team starts using GUI test automation as there is no way to tell if the selected testing strategy is correct without applying it first. Also, issues with maintenance can't arise without using and maintaining the tests.

Another interesting observation from the interviews was that the compiled list of benefits and challenges and limitations of GUI test automation were almost identical to the lists compiled based on the literature review in chapter 2.4. While no definite conclusions can be made based on this, it reinforces the credibility of the lists as they have no ties to each other. The other one represents the findings of previous research and the other a snapshot of the views and opinions of the industry practitioners.

7.3 Threats to validity and future work

The data on the benefits and cost-efficiency of the implemented test automation is very lacking as there is no data on its use and maintenance. Because of this, the focus of this thesis shifted to the challenges and impediments for adopting test automation and the research question on the benefits and cost-efficiency could only be partly answered. When the development teams eventually manage to adopt the use of GUI test automation in their development processes, the data on the reduced need for manual testing and other direct and perceived benefits should be gathered through collecting statistics on number of issues found and hours used. Also another round of interviews should be conducted and finally results compared with the costs of maintaining and using the automation.

The test automation plan designed as a part of the case study is intended to be the first step in adopting GUI test automation in the development teams. It only involves manually running the test automation as part of the development and testing processes. This solution requires the least changes in the current development processes and imposes the least amount of extra effort for the team. When the use of test automation is successfully adopted, the next steps would be to run the test automation as part of the team’s CI pipeline and look into using cloud-based device farms for maximum platform and device coverage for the tests.

A major part of the results and conclusions of this thesis is based on the opinions and views of the interviewed development team members. The lack of other evidence to back the opinions decreases the validity of the results and conclusions. However, all of the challenges and impediments encountered in the implementation of the test automation and listed by the interviewees were also identified in the literature review, which increases the validity of the results.

There are many approaches to GUI test automation as presented in chapter 2.2. The concept of GUI is also very wide as there are various different technologies and platforms for implementing GUIs such as various mobile platforms, websites and desktop applications. In this thesis the discussion of the benefits, cost-efficiency, challenges, limitations and impediments doesn’t take either of these into account by any measure. The majority of the existing research seems to be either about test automation or GUI test automation in general or focus on the characteristics of one specific approach. The literature used for the literature review was a mix of these two types of research articles and the articles on test automation in general mostly didn’t take these factors into account either.

It seems that further research on the subject of the usefulness and cost-efficiency of test automation is needed to establish better insight and industry knowledge on the best practices and typical pitfalls of different GUI testing techniques. GUI test automation is still a relatively little-used software development practice and there are contradicting results and opinions on using it, so clearer and more widely applicable knowledge on the subject could help the whole software industry to utilize GUI test automation more efficiently.

Bibliography

Alégroth, E., Feldt, R., 2017. On the long-term use of visual gui testing in industrial practice: a case study. Empir. Softw. Eng. 22, 2937–2971. https://doi.org/10.1007/s10664-016-9497-6

Alégroth, E., Feldt, R., Ryrholm, L., 2015. Visual GUI testing in practice: challenges, problemsand limitations. Empir. Softw. Eng. 20, 694–744. https://doi.org/10.1007/s10664-013-9293-5

Appium: Mobile App Automation Made Awesome. [WWW Document], 2019. URL http://appium.io/ (accessed 5.23.19).

Banerjee, I., Nguyen, B., Garousi, V., Memon, A., 2013. Graphical user interface (GUI) testing: Systematic mapping and repository. Inf. Softw. Technol. 55, 1679–1694. https://doi.org/10.1016/j.infsof.2013.03.004

Berner, S., Weber, R., Keller, R.K., 2005. Observations and Lessons Learned from Automated Testing, in: Proceedings of the 27th International Conference on Software Engineering, ICSE ’05. ACM, New York, NY, USA, pp. 571–579. https://doi.org/10.1145/1062455.1062556

Brooks, P.A., Robinson, B.P., Memon, A.M., 2009. An Initial Characterization of Industrial Graphical User Interface Systems, in: 2009 International Conference on Software Testing Verification and Validation. Presented at the 2009 International Conference on Software Testing Verification and Validation, pp. 11–20. https://doi.org/10.1109/ICST.2009.11

Carvalho, R.E.V. de, 2016. A Comparative Study of GUI Testing Approaches.

Dustin, E., 2002. Effective Software Testing: 50 Ways to Improve Your Software Testing. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Engström, E., Runeson, P., 2010. A Qualitative Survey of Regression Testing Practices, in: Product-Focused Software Process Improvement, Lecture Notes in Computer Science. Presented at the International Conference on Product Focused Software Process Improvement, Springer, Berlin, Heidelberg, pp. 3–16. https://doi.org/10.1007/978-3-642-13792-1_3

Garousi, V., Mesbah, A., Betin-Can, A., Mirshokraie, S., 2013. A systematic mapping study of web application testing. Inf. Softw. Technol. 55, 1374–1396. https://doi.org/10.1016/j.infsof.2013.02.006

GitHub - jpillora/xhook: Easily intercept and modify XHR request and response [WWW Document], 2019. URL https://github.com/jpillora/xhook (accessed 5.23.19).

Haugset, B., Hanssen, G.K., 2008. Automated Acceptance Testing: A Literature Review and an Industrial Case Study, in: Agile 2008 Conference. Presented at the Agile 2008 Conference, pp. 27–38. https://doi.org/10.1109/Agile.2008.82

Hevner, A., Chatterjee, S., 2010. Design Science Research in Information Systems, in: Design Research in Information Systems. Springer US, Boston, MA, pp. 9–22. https://doi.org/10.1007/978-1-4419-5653-8_2

Honkanen, H., 2016. Investigating Effects of Test Automation In a Large Software Project: A Case Study 72.

https://daringfireball.net/projects/markdown/ [WWW Document], 2019. URL https://daringfireball.net/projects/markdown/ (accessed 5.23.19).

https://github.com/nightwatchjs/nightwatch, 2019. . Nightwatch.js.

https://martinfowler.com [WWW Document], 2019. . martinfowler.com. URL https://martinfowler.com/bliki/PageObject.html (accessed 5.22.19).

http://usejsdoc.org/ [WWW Document], 2019. . usejsdoc.org. URL http://usejsdoc.org/ (accessed 5.22.19).

Kasurinen, J., Taipale, O., Smolander, K., 2010. Software Test Automation in Practice: Empirical Observations [WWW Document]. Adv. Softw. Eng. https://doi.org/10.1155/2010/620836

Leivo, T., 2017. Automating user interface testing: Case study at Finnish Transport Agency 61.

Leotta, M., Clerissi, D., Ricca, F., Spadaro, C., 2013a. Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops. Presented at the 2013 IEEE 6th International Conference On Software Testing, Verification and Validation Workshops (ICSTW), IEEE, Luxembourg, Luxembourg, pp. 108–113. https://doi.org/10.1109/ICSTW.2013.19

Leotta, M., Clerissi, D., Ricca, F., Tonella, P., 2014. Visual vs. DOM-Based Web Locators: An Empirical Study, in: Casteleyn, S., Rossi, G., Winckler, M. (Eds.), Web Engineering. Springer International Publishing, Cham, pp. 322–340. https://doi.org/10.1007/978-3-319-08245-5_19

Leotta, M., Clerissi, D., Ricca, F., Tonella, P., 2013b. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution, in: 2013 20th Working Conference on Reverse Engineering (WCRE). Presented at the 2013 20th Working Conference on Reverse Engineering (WCRE), IEEE, Koblenz, Germany, pp. 272–281. https://doi.org/10.1109/WCRE.2013.6671302

Liebel, G., Alégroth, E., Feldt, R., 2013. State-of-Practice in GUI-based System and Acceptance Testing: An Industrial Multiple-Case Study, in: 2013 39th Euromicro Conference on Software Engineering and Advanced Applications. Presented at the 2013 39th Euromicro Conference on Software Engineering and Advanced Applications, pp. 17–24. https://doi.org/10.1109/SEAA.2013.29

Liu, H., Kuan Tan, H.B., 2009. Covering code behavior on input validation in functional testing. Inf. Softw. Technol. 51, 546–553. https://doi.org/10.1016/j.infsof.2008.07.001

Model–view–controller, 2019. . Wikipedia.

Nidhra, S., 2012. Black Box and White Box Testing Techniques - A Literature Review. Int. J. Embed. Syst. Appl. 2, 29–50. https://doi.org/10.5121/ijesa.2012.2204

Rafi, D.M., Moses, K.R.K., Petersen, K., Mäntylä, M.V., 2012. Benefits and Limitations of Automated Software Testing: Systematic Literature Review and Practitioner Survey, in: Proceedings of the 7th International Workshop on Automation of Software Test, AST '12. IEEE Press, Piscataway, NJ, USA, pp. 36–42.

Selenium - Web Browser Automation [WWW Document], 2019. URL https://www.seleniumhq.org/ (accessed 5.23.19).

Sharma, M., Angmo, R., 2014. Web based Automation Testing and Tools.

Silva, J.L., Campos, J.C., Paiva, A.C.R., 2008. Model-based User Interface Testing With Spec Explorer and ConcurTaskTrees. Electron. Notes Theor. Comput. Sci. 208, 77–93. https://doi.org/10.1016/j.entcs.2008.03.108

Sivanandan, S., B, Y.C., 2014. Agile development cycle: Approach to design an effective Model Based Testing with Behaviour driven automation framework, in: 20th Annual International Conference on Advanced Computing and Communications (ADCOM). Presented at the 20th Annual International Conference on Advanced Computing and Communications (ADCOM), pp. 22–25. https://doi.org/10.1109/ADCOM.2014.7103243

The Apache Software Foundation, 2018. Architectural overview of Cordova platform - Apache Cordova [WWW Document]. URL

https://cordova.apache.org/docs/en/latest/guide/overview/index.html (accessed 8.2.18).
Vesikkala, M., 2014. Visual Regression Testing for Web Applications 78.
WebDriver [WWW Document], 2019. URL https://www.w3.org/TR/webdriver/ (accessed 5.23.19).
Whittaker, J.A., 2000. What is software testing? And why is it so hard? IEEE Softw. 17, 70–79. https://doi.org/10.1109/52.819971
Wiklund, K., Eldh, S., Sundmark, D., Lundqvist, K., 2017. Impediments for software test automation: A systematic literature review. Softw. Test. Verification Reliab. 27, e1639. https://doi.org/10.1002/stvr.1639